Hadoop SQL in a Blind Panic!

by Scott L. Hecht ~~Edna St. Vincent Millay~~

PUBLISHED BY CreateSpace, Inc. & Kindle Direct Publishing
Divisions of Amazon.com

ISBN-13: 9798831549027
Cover Image by **Sasimaporn Moonthep** (https://pixabay.com/users/_namfon_-16559457/)

Meet the Reviewers

Ramesh Mandalapu

Ramesh is Senior Principal Systems Administrator at ICON focusing on Cloudera Hadoop and Big Data. Prior to that, Ramesh worked for PRA Health Sciences, Symphony Health, The Hartford, Hamilton Sundstrand, Computer Associates and Tata Consultancy Services. Ramesh received a Bachelor of Science degree in Computer Science from Andhra University and a Master's Degree from Rensselaer Polytechnic Institute. Ramesh is a Cloudera Certified Professional (CCP).

Jordan Klein

Jordan is a Founder of Custom Data Shop, a company providing commercial solutions built primarily on federal health care data. Previously, Jordan worked for SDI, IQVIA, and Pfizer Vaccines. Jordan is a SAS Certified Base & Advanced Programmer with decades of experience with Patient-Level Data, Reference Data, and Master Data Management. Being involved in many M&A data integrations (SDI/Verispan/IQVIA/Cegedim Health/Quintiles), Jordan's expertise primarily pertains to provider mastering/matching/bridging, specialty mapping, and ETL coding/automation.

Contents at a Glance

Contents in Detail

Introduction

Why I Wrote This Book

You've been working at your job as a SQL programmer for quite a while now. You've become proficient in Oracle or SQL Server or Teradata, etc. You've coded oodles of SQL and things are running smoothly. Life's pretty sweet!

Then, your boss walks into your cubicle and announces your entire department is moving to Hadoop…and you have one month to convert everything over!

Your jaw drops! The contents of your intestines instantly liquefy! *Isn't a hadoop an animal only spotted on the plains of the Serengeti?* You go into a blind panic! **PANIC, I say!! PANIC!!!**

But, don't worry, that's where this book comes in. My department went through the same thing a few years back and in order to save you time and frustration, I wrote down what we learned so you don't have to go through the same **PANIC – PANIC, I say!! PANIC!!!** – as we did. Whether you can learn anything from my notes remains to be seen…I await the tarring and feathering!

This book will teach you how to work with Impala SQL, HPL/SQL, the Linux operating system, Hadoop/HDFS commands, and much more. With that said, this book won't teach you how to become a Hadoop Administrator; that's not its focus. Besides, there are already several excellent books written on that topic.

By the end of the book, you should be able to transfer all (or most) of your tables from your legacy database into Hadoop as well as set up automated scripts/processes to keep these tables updated. You should then be able to quickly move your entire programming team over to Hadoop without too much trouble…I can't do anything about their endless complainin', though!

So, what are you waiting for? You only have a bloody month! QUICK! START READING!!

Organization of This Book

This book is broken up into nine parts:

In *Part I - Getting Starting*, we take a quick tour on how to pull data from your legacy database into Hadoop, load a delimited file into the Hadoop database and how to work with some Hadoop commands in *Chapter 1 – Quick Start Guide*. Don't worry! Everything you see in *Chapter 1 – Quick Start Guide* will be explained in excruciating detail throughout the book.

We then craft a detailed e-mail in *Chapter 2 – Hadoop Administrator E-Mail* which you'll send to your Hadoop Administrator to obtain all (or most) of the information you'll need to start working with Hadoop immediately. Rather than constantly e-mailing your Hadoop Administrator over and over again, you can take care of it in one *swell foop*! While much of the information you receive back in response may look like incantations to summon the hounds of hell, we'll make use of this important information in subsequent chapters.

In *Chapter 3 – Recommended Windows Client Software*, I recommend software which you can use to interact with the Hadoop database, whether running SQL queries or accessing the Linux command line to run scripts. And in *Chapter 4 – A Teensy-Weensy Chat about Hadoop*, we chat about what Hadoop is and how you should think about it…all without, hopefully, melting your eyeballs. In *Chapter 5 – Creating Your Very Own Hadoop Playground*, we round out *Part I - Getting Starting* by describing how to create your own Hadoop database on your Windows laptop.

In *Part II - Querying the Hadoop Database*, we start off with a brief introduction to generic SQL in *Chapter 6 – Introduction to SQL*. We then show you several ways to access the Hadoop database either through its web interface *Hue* or via a dedicated Windows SQL client application in *Chapter 7 – Querying the Hadoop Database (Hue and SQL Clients)*. We then discuss ImpalaSQL data definition language (DDL) and data manipulation language (DML) in *Chapter 8 – The One About ImpalaSQL*, the SQL flavor we'll use throughout the book. In

Chapter 9 – ImpalaSQL Functions Parade, we describe most of the aggregate and single-row functions available in ImpalaSQL in detail. Since date and time functions can be a nightmare to use, we dedicate *Chapter 10 – Voyage of the Damned (Dates & Times – ImpalaSQL Edition)* to discussing these functions in particular. In *Chapter 11 – Regular Expressions*, we discuss regular expressions and how to use them in ImpalaSQL. ImpalaSQL allows you to make use of analytic (windowing) functions, so we discuss these in detail in *Chapter 12 – SQL Analytic (Windowing) Functions*. Although not necessarily available in ImpalaSQL yet, we discuss extensions to the GROUP BY Clause in *Chapter 13 – Extensions to the GROUP BY Clause*. Complex data types allow you to extend your usual SQL thinking with arrays, maps, structures, etc., and these are discussed in *Chapter 15 – Complex Data Types in HiveQL and ImpalaSQL*. Occasionally, one of your SQL queries will run like a sloth on crutches, so in *Chapter 16 – SQL Performance Improvements*, we outline several methods you can use to get your queries running at warp speed.

In *Part III - Working with the Linux Operating System*, we show you how to connect to your Linux edge node server in *Chapter 17 – PuTTY and the Linux Edge Node Server*. In *Chapter 18 – Introduction to the Linux Operating System*, we discuss how to use the Linux operating system in detail. Since a text editor is your friend, we show you how to use the vi Editor in *Chapter 19 – Introduction to the vi Editor*. In *Chapter 20 – Working with Linux Scripts*, we show you how to create and execute scripts in Linux. Finally, we show you how to run ImpalaSQL from the Linux command line in *Chapter 21 – Running ImpalaSQL from the Linux Command Line*.

In *Part IV - Working with Hadoop*, we explain how to work with Hadoop commands from the Linux command line to interact directly with Hadoop from Linux (*Chapter 22 – Hadoop Commands from Linux (hadoop/hdfs)*). In *Chapter 23 – Working with Managed and External Tables*, we go into detail on how to work with managed and external tables as well as how to export table data to delimited files. In *Chapter 24 – The Impala Queries Webpages*, we briefly discuss how to use the Impala queries web pages to kill misbehaving SQL queries.

In *Part V - HPL/SQL Procedural Language*, we show you how to use the Hadoop procedural language HPL/SQL. This is similar to Oracle's PL/SQL and SQL Server's T-SQL. Because, you know, not everything can be done just in SQL! Surely, you jest! I do not jest…and don't call me Shirley. In *Chapter 25 – Introduction to HPL/SQL*, we start off with a basic introduction to HPL/SQL. We then move on to using HPL/SQL to interact with a database in *Chapter 27 – HPL/SQL and Chatting with a Database*. In *Chapter 28 – Handling HPL/SQL Exceptions*, we discuss how to handle errors that may pop up in your HPL/SQL code.

In *Part VI - Updating Your Database*, we show you how to use `sqoop` to import table data from your legacy database into the Hadoop database as well as export table data directly to your current database (*Chapter 29 – Database Import/Export Using sqoop*). Besides `sqoop`, you can use the `LOAD DATA` Statement to load data directly into the database and we discuss this in *Chapter 30 – Loading Data using LOAD DATA to Load Data*. To make your life easier, you can run your Linux scripts anytime by scheduling them with the Linux built-in scheduler cron and we discuss this in detail in *Chapter 31 – Scheduling Jobs Using crontab*. In *Chapter 32 – Updating Your Hadoop Tables with make*, we discuss how to use the Linux utility `make` to help load/update your database tables easier.

In *Part VII - Advanced Topics I*, we blow your socks off by discussing the Hadoop MetaStore (*Chapter 33 – Accessing the Hive MetaStore*), Impala Resource Pools (*Chapter 34 – Working with Impala Request Pools*); making backup copies of your directories using `tar` (*Chapter 35 – Making a Backup Copy of a Linux Directory*); running Linux commands using `ssh` as well as transferring files using `scp` (*Chapter 36 – Using ssh and scp from Linux and Windows*). In *Chapter 37 – The Linux /etc/skel Directory*, we discuss how to use the Linux `/etc/skel` directory to make setting up a new Linux user easier. Finally, in *Chapter 38 – The parquet-tools and parquet-cli Utilities*, we discuss the Linux utilities `parquet-tools` and `parquet-cli` both of which allow you to view a file stored in the Parquet format.

In *Part VIII - Advanced Topics II*, we show you how to create your own ImpalaSQL functions in Java. We start off with a basic introduction to Java programming (*Chapter 39 – Quick Start Guide to Java Programming*) and then discuss how to create your very own user-defined function in *Chapter 40 – Creating User-Defined Functions (UDFs) for ImpalaSQL*.

In *Part IX - Appendages*, we display the entire Hadoop Administrator E-Mail in *Appendage #1 – Hadoop Administrator E-Mail*. In *Appendage #2 – Linux on Windows*, we discuss how to setup and run Linux on Windows. In *Appendage #3 – When HPL/SQL Causes You Pain*, we discuss how to set up HPL/SQL, if it's

causing you trouble. We discuss several common error messages in *Appendage #4 – When Bad Errors Happen to Good Programmers*. In *Appendage #5 – Where Do I Go from Here?*, we suggest further reading. Finally, since the book is way too thin, the ISO Latin-1 character set is included in *Appendage #6 – ISO Latin-1 (8859-1) Character Set*.

Support

You can find the Hadoop Administrator e-mail, in all its black-and-white glory, on my personal website at `www.sheepsqueezers.com` (…I know, I know, it's a silly name…) along with much of the code shown throughout the book. You can also e-mail me at `blindpanic@sheepsqueezers.com`.

Caution and Warning

Throughout the book, I refer to your current database – whether Oracle, SQL Server, Teradata, etc. – as your *legacy* database. I thought this was more respectful than *wizened-old-codger* database.

I avoid using the word *subdirectory* and instead just use the word *directory* throughout. It saves on printing costs. AH HA HA HA HA!! Clearly, any directory created under any other directory is a subdirectory.

In the examples shown below, I refer to both your legacy database schema and Hadoop database schema as `prod_schema`, the Hadoop database as `hdpserver` and the Linux edge node server as `lnxserver`. These must be replaced with the correct schema, database and Linux server host names given to you by upstanding Hadoop and Linux Administrators.

Also, I refer to your Linux home directory as `/home/smithbob` as well as your team's HDFS working directory as `/data/prod/teams/prod_schema`. Naturally, you'll have to replace these with the directory names given to you by your Hadoop and Linux administrators.

As indicated above, I'll be focusing on the Impala query engine, but will comment on the Hive query engine often. While you may initially think that there's a *Hatfields vs. McCoys* battle raging between them, but nothing is farther from the truth, as we discuss later in the book.

While I would love to say this book is the *be all and end all* of the Hadoop SQL world, that would be a massive lie. The way I do things may not be the most efficient, but they're easy to understand, less mysterious than other methods I've seen, and just plain work for me…and I hope work for you, too. Note that this book was created using the Cloudera version of Hadoop and due to differences in Hadoop flavors as well as their own internal versions, some of the commands may not work exactly as advertised. Please work with your superhero Hadoop Administrator to resolve any issues.

Finally, I would love to hear about your Hadoop successes, so please feel free to e-mail me at `blindpanic@sheepsqueezers.com`. If you've found any issues with the book, would like to have something added or think something needs to be changed, please feel free to e-mail me and I'll try to incorporate your comments into the second edition of the book. Thanks!!

PART I - *Getting Started*

Chapter 1 – Quick Start Guide

In this chapter, we walk through a series of typical steps you might perform on a regular basis in the Hadoop database including downloading data from your legacy database, loading text files into the Hadoop database directly and working with Hadoop commands to create a delimited export file from a Hadoop table. Note that I've eliminated many of the parameters in the commands below in order to keep the text clean. We discuss these commands and their parameters in subsequent chapters in eye-watering detail.

Let's assume that you have a dimension table named `DIM_POSTAL_CODE` in your legacy database schema `prod_schema` containing the following columns:

COLUMN NAME	DATA TYPE
POSTAL_CODE	VARCHAR(5)
CITY	VARCHAR(50)
STATE_CODE	VARCHAR(3)
LATITUDE	NUMBER
LONGITUDE	NUMBER

Below, we'll use the Linux command line utility `sqoop` to pull the data in this table down to a temporary table in the Hadoop database. Since `sqoop` may create this table with numeric data stored as strings, we'll insert this data into a final table with the appropriate data types using the `CAST` function.

To generate the SQL code for the final table (semi-) automatically, you can use your legacy database's metadata tables such as `ALL_TAB_COLUMNS` in Oracle, `INFORMATION_SCHEMA.COLUMNS` in SQL Server, `DBC.COLUMNS` in Teradata, and so on. Below is SQL code you can use as a starting point, but you'll have to modify it for your metadata tables, data types, special conditions, etc.:

```
WITH vwMETA AS (
                SELECT LOWER(TABLE_NAME) AS TABLE_NAME,
                       LOWER(COLUMN_NAME) AS COLUMN_NAME,
                       COLUMN_ID AS COLUMN_ID,
                       LOWER(DATA_TYPE) AS DATA_TYPE,
                       ROW_NUMBER() OVER (PARTITION BY TABLE_NAME ORDER BY COLUMN_ID) AS RNBR,
                       COUNT(*) OVER (PARTITION BY TABLE_NAME) AS TOT_RNBR
                 FROM ALL_TAB_COLUMNS
                 WHERE OWNER='PROD_SCHEMA'
                       AND TABLE_NAME IN ('DIM_POSTAL_CODE')
               )
SELECT TABLE_NAME,
       LISTAGG(CT,' ') WITHIN GROUP (ORDER BY RNBR) AS CT_FINAL
 FROM (
       SELECT TABLE_NAME,
              RNBR,
              CASE
               WHEN RNBR=1 AND RNBR<>TOT_RNBR
                THEN 'create table prod_schema.' || TABLE_NAME || '(' || COLUMN_NAME || ' ' || DT || ','
               WHEN RNBR=1 AND RNBR=TOT_RNBR
                THEN 'create table prod_schema.' || TABLE_NAME || '(' || COLUMN_NAME || ' ' || DT
               WHEN RNBR>1 AND RNBR<>TOT_RNBR
                THEN COLUMN_NAME || ' ' || DT || ','
               WHEN RNBR>1 AND RNBR=TOT_RNBR
                THEN COLUMN_NAME || ' ' || DT || ');'
              END AS CT
        FROM (
              SELECT COLUMN_ID,TABLE_NAME,COLUMN_NAME,RNBR,TOT_RNBR,
                     CASE
                      WHEN DATA_TYPE IN ('char','varchar','varchar2') THEN 'string'
                      WHEN DATA_TYPE='date' THEN 'timestamp'
                      WHEN DATA_TYPE='number' AND COLUMN_NAME IN ('latitude','longitude') THEN 'double'
                      WHEN DATA_TYPE='number' THEN 'bigint'
                      ELSE '?????'
                     END AS DT
               FROM vwMETA
              )
      )
GROUP BY TABLE_NAME
ORDER BY TABLE_NAME;
```

The result of running this code for the table `DIM_POSTAL_CODE` on the legacy database is as follows (indentation provided for clarity):

```
create table prod_schema.dim_postal_code(postal_code string,
                                         city string,
                                         state_code string,
                                         latitude double,
                                         longitude double);
```

Now, let's use the command line utility `sqoop` from the Linux command line. In order to gain access to the Linux command line from your Windows laptop, you use an application called PuTTY to connect to the Linux server. This application provides you with a command line used to interact directly with the Linux server. You can think of the Linux command line similar to the Windows Command Prompt. When you log into the Linux server, you're placed into your own account's home directory where you can create Python programs, Linux scripts, store data, etc. We describe PuTTY in much more detail later in the book. We use `sqoop` to copy the table from the legacy database into a temporary table in the Hadoop database (the **abridged** code below should be placed on a single line…my OCD forces me to line things up like this…I'm taking medication for it…it's not working…):

```
sqoop import --hive-table TMP_DIM_POSTAL_CODE
             --connect <legacy_db_connection_info>
             --table DIM_POSTAL_CODE
             --target-dir /data/prod/teams/prod_schema/TMP_DIM_POSTAL_CODE
```

After a long series of cryptic messages flying lightning fast across the screen, your table is downloaded into the table `TMP_DIM_POSTAL_CODE` in your Hadoop schema `prod_schema`. Let's look at the first five rows using the ImpalaSQL command line query utility `impala-shell` which you start from the Linux command line:

```
[hdpserver:21000] prod_schema> select *
                                from prod_schema.tmp_dim_postal_code
                                order by 1
                                limit 5;
+-------------+------------+------------+-----------+------------+
| postal_code | city       | state_code | latitude  | longitude  |
+-------------+------------+------------+-----------+------------+
| 00210       | PORTSMOUTH | NH         | 43.005895 | -71.013202 |
| 00211       | PORTSMOUTH | NH         | 43.005895 | -71.013202 |
| 00212       | PORTSMOUTH | NH         | 43.005895 | -71.013202 |
| 00213       | PORTSMOUTH | NH         | 43.005895 | -71.013202 |
| 00214       | PORTSMOUTH | NH         | 43.005895 | -71.013202 |
+-------------+------------+------------+-----------+------------+
```

Unfortunately, `sqoop` may convert some data types to strings (except for dates/times…we talk more about this later in the book). The proof for our table above is shown below by describing the table:

```
[hdpserver:21000] prod_schema> desc tmp_dim_postal_code;
+-------------+--------+---------+
| name        | type   | comment |
+-------------+--------+---------+
| postal_code | string |         |
| city        | string |         |
| state_code  | string |         |
| latitude    | string |         |
| longitude   | string |         |
+-------------+--------+---------+
```

Let's take control of this dastardly situation and create our final dimension table with the appropriate data types using ImpalaSQL.

First, let's delete our final table, if it already exists (`purge` prevents the table from being stored temporarily in the recycle bin):

```
[hdpserver:21000] prod_schema> drop table if exists
                                    prod_schema.dim_postal_code purge;
```

Next, let's recreate our final table using the `CREATE TABLE` Statement we generated in the legacy database:

```
[hdpserver:21000] prod_schema> create table prod_schema.dim_postal_code(
                                              postal_code string,
                                              city string,
                                              state_code string,
                                              latitude double,
                                              longitude double);
```

Finally, let's insert the data from the temporary table into our final table by converting the columns `latitude` and `longitude` to the `double` data type using the `CAST` function. The other columns remain the same data type: `string`.

```
[hdpserver:21000] prod_schema> insert into prod_schema.dim_postal_code
                                select postal_code,
                                       city,
                                       state_code,
                                       cast(latitude as double) as latitude,
                                       cast(longitude as double) as longitude
                                 from prod_schema.tmp_dim_postal_code;
```

When we describe our final table, you'll see that both `latitude` and `longitude` are `double`s:

```
[hdpserver:21000] prod_schema> desc prod_schema.dim_postal_code;
+-------------+--------+---------+
| name        | type   | comment |
+-------------+--------+---------+
| postal_code | string |         |
| city        | string |         |
| state_code  | string |         |
| latitude    | double |         |
| longitude   | double |         |
+-------------+--------+---------+
```

Finally, we can remove the temporary table to save space:

```
[hdpserver:21000] prod_schema> drop table if exists
                                  prod_schema.tmp_dim_postal_code purge;
```

Since our brand-spanking new Hadoop table contains the two-letter US state code in the column `state_code`, let's create another dimension table mapping from two-letter US state code to the associated US state name. We download this file from the Internet – after watching some fan-made Star Trek videos on YouTube, checking a bid on eBay, and having a jolly ol' laugh at AliExpress – and call it `us_state_mapping.csv`. This file contains two columns `state_code` and `state_name` as well as a single header row. Let's load this comma-delimited file into the Hadoop database directly, completely bypassing our adult-diaper-wearing…er…legacy database. Here are the first few rows from the text file:

```
state_code,state_name
aa,u.s. armed forces - americas
ae,u.s. armed forces - europe
ak,alaska
al,alabama
ap,u.s. armed forces - pacific
```

```
ar,arkansas
as,american samoa
az,arizona
ca,california
co,colorado
...snip...
```

Currently, this file resides on your company laptop…and that ain't gonna do you no good! You'll need to use a file transfer program to copy this file over to your account on the Linux server. We discuss FTP software further below, but let's assume you FTP'd the file `us_state_mapping.csv` to your Linux home directory, say, `/home/smithbob`.

Next, we need to copy the file `us_state_mapping.csv` from your Linux home directory `/home/smithbob` to an appropriate directory under your team's Hadoop directory `/data/prod/teams/prod_schema`. If we skip this step, Hadoop won't know about your file, we won't be able to load it into the Hadoop database, and war will break out in Denmark. Let's create a directory under `/data/prod/teams/prod_schema` named, say, `tmp_us_state_mapping` using the following Hadoop command from the Linux command line:

```
hadoop fs -mkdir /data/prod/teams/prod_schema/tmp_us_state_mapping
```

Next, let's copy the file `us_state_mapping.csv` from our Linux directory `/home/smithbob` to a file named, say, `tmp_us_state_mapping.csv` into our new Hadoop folder `/data/prod/teams/prod_schema/tmp_us_state_mapping`. From the Linux command line, issue the following command (on one line):

```
hadoop fs -copyFromLocal
     /home/smithbob/us_state_mapping.csv
     /data/prod/teams/prod_schema/tmp_us_state_mapping/tmp_us_state_mapping.csv
```

Next, let's check our Hadoop directory to ensure the file is actually there. From the Linux command line, issue the following command:

```
hadoop fs -ls -R /data/prod/teams/prod_schema/tmp_us_state_mapping
```

The output from this command is shown below confirming the existence of our file:

```
-rw-r-----   3 smithbob teamgroup        989 2021-09-09 12:36
     /data/prod/teams/prod_schema/tmp_us_state_mapping/tmp_us_state_mapping.csv
```

Now that the file is located in a directory of Hadoop, we can use ImpalaSQL to create SQL code used to access the data in `tmp_us_state_mapping.csv` and then create our final dimension table `dim_us_state_mapping`. First, let's delete the temporary table just in case it already exists:

```
[hdpserver:21000] prod_schema> drop table if exists
                                   prod_schema.tmp_us_state_mapping purge;
```

Next, let's access that data using the `CREATE EXTERNAL TABLE` Statement providing the location of our data, **not the name of the file itself**, on the `LOCATION` Clause:

```
[hdpserver:21000] prod_schema>
create external table prod_schema.tmp_us_state_mapping(state_code string,
                                                       state_name string)
 row format delimited
 fields terminated by ','
 stored as textfile
 location '/data/prod/teams/prod_schema/tmp_us_state_mapping'
 tblproperties('skip.header.line.count'='1');
```

We indicate that we want to skip the header row by providing the `skip.header.line.count` table properties option and set it to a value of `1`.

In ImpalaSQL, let's take a look at a few rows of data in the external table `tmp_us_state_mapping`:

```
[hdpserver:21000] prod_schema> select *
                                from prod_schema.tmp_us_state_mapping
                                limit 10;
+------------+-------------------------------+
| state_code | state_name                    |
+------------+-------------------------------+
| aa         | u.s. armed forces - americas |
| ae         | u.s. armed forces - europe   |
| ak         | alaska                        |
| al         | alabama                       |
| ap         | u.s. armed forces - pacific  |
| ar         | arkansas                      |
| as         | american samoa                |
| az         | arizona                       |
| ca         | california                    |
| co         | colorado                      |
+------------+-------------------------------+
```

Next, let's create our final table to hold the data. Note that we're trimming off the blanks using the `TRIM` function as well as uppercasing `state_code` and `state_name` using the `UPPER` function:

```
[hdpserver:21000] prod_schema>
drop table if exists prod_schema.dim_us_state_mapping purge;

create table prod_schema.dim_us_state_mapping(state_code string,
                                              state_name string);

insert into prod_schema.dim_us_state_mapping
 select upper(trim(state_code)) as state_code,
        upper(trim(state_name)) as state_name
  from prod_schema.tmp_us_state_mapping;
```

Finally, let's drop the temporary table to save space:

```
[hdpserver:21000] prod_schema> drop table if exists
                                        prod_schema.tmp_us_state_mapping purge;
```

And, let's look at a few rows from our final dimension table `prod_schdema.dim_us_state_mapping`:

```
[hdpserver:21000] prod_schema> select *
                                from prod_schema.dim_us_state_mapping
                                limit 10;
+------------+-------------------------------+
| state_code | state_name                    |
+------------+-------------------------------+
| AA         | U.S. ARMED FORCES - AMERICAS |
| AE         | U.S. ARMED FORCES - EUROPE   |
| AK         | ALASKA                        |
| AL         | ALABAMA                       |
| AP         | U.S. ARMED FORCES - PACIFIC  |
| AR         | ARKANSAS                      |
| AS         | AMERICAN SAMOA                |
| AZ         | ARIZONA                       |
| CA         | CALIFORNIA                    |
| CO         | COLORADO                      |
+------------+-------------------------------+
```

Now, one of our colleagues, Big Mike the Sales Guy, wants an Excel spreadsheet created based on the join of the two tables `dim_postal_code` and `dim_us_state_mapping`. Let's create a tab-delimited text file for him by using the `CREATE EXTERNAL TABLE` Statement along with an appropriate ImpalaSQL query joining both tables together by the `state_code` column. First, let's create our output table for Big Mike the Sales Tramp. You'll note that this is similar to what we just did above to read in our text file:

```
[hdpserver:21000] prod_schema>
drop table if exists prod_schema.bigmike_output purge;
create external table prod_schema.bigmike_output(postal_code string,
                                                  city string,
                                                  state_code string,
                                                  latitude double,
                                                  longitude double,
                                                  state_name string)
 row format delimited
  fields terminated by '\t'
 stored as textfile
 tblproperties('serialization.null.format'='');
```

Using the `INSERT` Statement, let's insert the results of the query below into the table `bigmike_output`:

```
[hdpserver:21000] prod_schema>
insert into prod_schema.bigmike_output
 select A.postal_code,
        A.city,
        A.state_code,
        A.latitude,
        A.longitude,
        B.state_name
  from prod_schema.dim_postal_code A
   left join prod_schema.dim_us_state_mapping B
   on A.state_code=B.state_code;
```

At this point, our table `bigmike_output` is in Hadoop waiting for us to make the next move. In order to pull it out of Hadoop and into our Linux directory `/home/smithbob`, we use the Hadoop command `getmerge` from the Linux command line providing the Hadoop directory as well as the output location and file name:

```
hadoop fs -getmerge /data/prod/teams/prod_schema/bigmike_output
                    /home/smithbob/bigmike_output.tsv
```

When the command above completes, the file `/home/smithbob/bigmike_output.tsv` is ready to give to Big Mike the Sales Strumpet. Here are a few rows from the tab-delimited file:

```
00623   CABO ROJO      PR      18.08643        -67.15222        PUERTO RICO
00633   CAYEY          PR      18.194527       -66.1834669      PUERTO RICO
00640   COAMO          PR      18.077197       -66.359104       PUERTO RICO
00676   MOCA           PR      18.37956        -67.0842399      PUERTO RICO
00728   PONCE          PR      18.013353       -66.65218        PUERTO RICO
00734   PONCE          PR      17.999499       -66.643934       PUERTO RICO
00735   CEIBA          PR      18.258444       -65.65987        PUERTO RICO
00748   FAJARDO        PR      18.326732       -65.652484       PUERTO RICO
00766   VILLALBA       PR      18.126023       -66.48208        PUERTO RICO
00771   LAS PIEDRAS    PR      18.18744        -65.87088        PUERTO RICO
```

Chapter 2 – Hadoop Administrator E-Mail

Before we begin doing anything, we'll need quite a few pieces of information only a Hadoop Administrator can give you. In this chapter, we draft an e-mail which you can send to your delightful Hadoop Administrator gathering all of the sordid details (i.e., useful information) you'll need up-front.

Now, I know what you're thinking: *Who's my Hadoop Administrator? Do we even have one?* Ah! You need to find that out uber-pronto. Send the following e-mail to your legacy Database Administrator e-mail group as well as any other e-mail groups you think appropriate. By the way, please feel free to change one minor aspect my e-mails: THE WORDS.

```
Database Administrators:

As you may be aware by now, the <insert dept name here> department has been
asked to move off the <insert legacy database name> database to the new Hadoop
database.  I don't want to do this, but Corporate has my kids. :-)

Can you please recommend either a Hadoop Administrator I can work with for the
duration of this conversion, or let me know whom I may contact to find out
this information?

Thanks,
Bob Smith
822-6235
smithbob@company.com
```

Once you determine whom to contact, you can send him/her the big honkin' e-mail below. I'll show you the e-mail in pieces so I can explain each part, but the full e-mail appears in *Appendage #1 – Hadoop Administrator E-Mail*.

```
Hadoop Administrators:

Tally Ho!  My name is Bob Smith and I work for the <insert dept name here>
department and, as you may have heard, I've been tasked with moving data off
our legacy <insert legacy database name> database to the Hadoop database.  I
was hoping that you could be my contact for the duration of this conversion.

First, thank you up-front for helping out since this Hadoop shizz is new to me
and my team.

Second, you probably won't be surprised that I have about a bazillion
questions for you which I've placed below.  Your responses will go a long way
in helping me and my team move to Hadoop as quickly (and painlessly!) as
possible.

Here goes...
```

It's always nice to start off an e-mail with pleasantries. (Send your wonderful Hadoop Administrator a lovely gift basket once the conversion is done!)

```
☐ Do you have a Linux edge node server that my team can use?  If so, what's
  the server's host name?  My team and I will be automating some processes
  using Linux scripts, so access to a Linux edge node server will help us out
  greatly.
```

An *edge node server* is a Linux server that's not one of the Hadoop servers used to process SQL queries, but rather allows access to Hadoop features such as those `hadoop` commands shown in *Chapter 1 – Quick Start Guide*, the ImpalaSQL shell used to query the database, and more. It allows users access to the database, run scripts (Linux,

Python, R, etc.), and will make your life a helluva lot easier than running everything from your company laptop all the time.

☐ My team and I plan to use PuTTY to connect to the Linux edge node server. I just want to confirm that we must use port 22 (SSH) when setting up a connection to the edge node server. Do you recommend something other than PuTTY?

PuTTY is one application you can use to access the Linux command line on the edge node server. Normally, you connect using a secure connection called SSH on the default SSH port `22`. The older connection is called Telnet, which is insecure and, if you use it, war will break out in Hungary.

☐ On our legacy database, the schema we use is named *<insert name of legacy database schema name here>*. Can you please set up the same schema name on the Hadoop database?

Having your Hadoop Administrator create the same schema name as you use in your legacy database may, I hope, reduce down the number of SQL code changes necessary during the conversion.

☐ Since my team and I will use the edge node server as well as the Hadoop database, can you please set up the following individuals with an account on the Linux edge node server as well as access to the Hadoop database schema requested above? *<insert your Team's corporate e-mail addresses here>*

Also, the following team members should be given privileged access to run Hadoop commands via hadoop/hdfs from the Linux command line: *<insert select team members who should have higher privileges, including yourself, here>*

Each member of your team will be given a separate Linux account that he/she can log into. In the examples above, the user `smithbob` has a Linux home directory named `/home/smithbob`. This will be individualized for each member of your team. Also, access to the schema is necessary, of course, just like for your legacy database. But, not every user needs to interact with Hadoop directly, as you saw in *Chapter 1 – Quick Start Guide*, using the `hadoop` command.

☐ Not all of my team members are highly technical, but would like to run simple queries against the Hadoop database. Do you have the Hadoop database web interface **Hue** set up and accessible? If so, what's the URL?

While the more technical members of your team may enjoy using PuTTY to interact with the Hadoop database from the Linux command line as well as be comfortable using a full-blown SQL client, some of your users will plummet to their demise if asked to jump across this particular technical chasm. Hue allows users to query the database from a web browser, so if they can use Facebook or buy toothpaste from Amazon, then *we all gud*.

☐ In order to kill runaway SQL queries, can you please list the URLs to the Hadoop query webpages? I believe these URLs generally use port 25000 (/queries), but don't hold me to that...I'm new to these parts.

Occasionally, a SQL query will try to take over the universe and your attempts to kill it from your SQL client's interface will fail miserably. Hadoop SQL is not immune to this. Rather than sending an e-mail to the Hadoop Administrator asking for the query to be killed, Hadoop has a series of webpages which list all currently running queries. You can locate the unruly query and click the Cancel button. *Boom! Dead! Done!*

☐ `Can you recommend a SQL client application (such as Toad Data Point, DBeaver, SQuirreL, etc.) for use with Hadoop? What do you use?`

There are several SQL clients available for you to use (some free and some with big honkin' price tags), but not all of them can communicate with Hadoop. For example, Oracle PL/SQL Developer only talks to Oracle. But, Oracle SQL Developer can talk to Hadoop (a bit…I'll explain later). Your Hadoop Administrator may have one or more suggestions for you, your IT Department may dictate the software you may use, or it may be up to you and your team to decide which application(s) to use. We give several examples later in the book.

☐ `Do you have Hive and Impala ODBC (32-bit/64-bit) and JDBC drivers available on the corporate network? If so, I'd like to access them so that I may set up my team's SQL client software (among other things). If not, can you recommend where I may download these drivers?`

No matter which SQL client you decide to use, you'll need the ODBC and/or JDBC drivers to allow the application to communicate with the Hadoop database. And which driver you use depends on the SQL client you're setting up. For example, Toad Data Point uses ODBC drivers, but DBeaver uses JDBC drivers. If you have to download the ODBC and JDBC drivers yourself, don't forget to select the 64-bit option instead of the 32-bit option unless you have a specific software application that requires the 32-bit driver. For example, the version of Microsoft Excel you have installed may, in fact, be 32-bit, so the 32-bit ODBC driver should be used to communicate with the Hadoop database from Excel, if that's a route you want to explore.

☐ `Speaking of ODBC and JDBC drivers, can you please provide example connection information/strings for both ODBC and JDBC connections to Hive (port 10000?) as well as Impala (port 21050?)? We'll be using the ODBC connection information with applications such as Microsoft Excel, PowerBI, Tableau, etc. The JDBC connection strings will be used with client software that uses JDBC rather than ODBC such as DBeaver, SQuirreL, etc.`

Having the ODBC and JDBC drivers available is one thing, but you need to know the connection information in order to allow your applications to communicate with the Hadoop database. With ODBC, not only can you set up Windows-based SQL clients, but you can query the Hadoop database directly from Microsoft Excel or within an application you create using Visual Studio as well as set up ODBC Data Source Names (DSNs). Pretty spiffy, huh?

☐ `Does our corporate network run Kerberos? If so, when creating cron jobs to run automatically, we may need to create a keytab file containing Kerberos-related information. Which encryption types do you suggest we include in the keytab file? arcfour-hmac-md5? aes256-cts? rc4-hmac? Any others? Also, what's our` **`Kerberos Realm`** `and` **`Host FQDN`**`? If not Kerberos, then LDAP?`

If you'd like to execute Linux scripts automatically anytime of the day or night, one utility available on Linux you can use is called `cron`. You edit a file called `crontab` and insert into it a list of script names you want to execute and specify when you want them to run (date, time, etc.). At that point, you can just go home and enjoy your life! If your company uses the Kerberos computer-network authentication protocol, and your Hadoop cluster has been *kerberized*, you'll need to provide additional information in the `crontab` file; otherwise, your jobs will not run at all. We discuss this more in *Chapter 31 – Scheduling Jobs Using crontab*. If you don't know anything about Kerberos, check out the excellent **F5 DevCentral** channel on YouTube to learn more. It's fascinating! And I'm using the word fascinating quite wrongly.

☐ `We would like the ability to access our legacy database (`*`<insert name of legacy database>`*`) from the Linux edge node server for use with sqoop and other tools. Can you please install the software necessary so that my team and I may access the legacy database from there?`

Despite calling your legacy database…uh…your legacy database, it may not actually be going away any time soon, if at all. Now, it might be useful to be able to query your legacy database from the Linux edge node server itself. For example, if your legacy database is Oracle, the Hadoop Administrator, in coordination with the Linux Administrator, can install Oracle's SQL*Net as well as SQL*Plus and SQL*Loader on the Linux edge node server. This will allow you to *query from*/*load to* Oracle from the Linux command line.

☐ Is there a generic account available on the Linux edge node server for me and a few of my team members to use? We'd like a single account to execute our production code. If so, can you please forward the username and password? If not, can you please create an account on the Linux edge node server whose password is static? Also, please give this account access to the appropriate schemas as well as hadoop/hdfs privileges.

As indicated above, rather than running scripts from each team member's Linux account, it's probably a good idea to have your code execute from a generic account. This is also useful if you plan to create an internal website accessible by the no-neck employees outside of your fab department. These no-necks can push a few buttons on a webpage, queries can be automatically kicked off on the Linux edge node server and the results can be lovingly e-mailed to them. Now that's a win! Do I hear pay raise!?! I think I do!!

☐ Is HPL/SQL available from the Linux edge node server? If not, can you please install it so that my team and I can create and execute procedures on the Linux edge node server against the Hadoop database? Also, where is the file hplsql-site.xml located?

As mentioned earlier, HPL/SQL is similar to Oracle's PL/SQL and SQL Server's T-SQL procedural languages. HPL/SQL is run using the `hplsql` utility from the Linux command line instead of directly in the database (like Oracle). The file `hplsql-site.xml` contains empty connections to Hive, Impala, etc. You'll have to work with your Hadoop Administrator to alter this file. We talk more about this when we discuss HPL/SQL in PART V, *HPLSQL*. We also discuss setting up HPL/SQL to execute properly in *Appendage #3 – When HPL/SQL Causes You Pain*.

☐ Is there a directory on the Linux edge node server where we can store the team's production code? If not, can you please create a directory accessible by my team as well as the generic account?

Rather than storing all production code (such as HPL/SQL programs) in a user's Linux account (even the generic account), all of your code should be stored in a directory on the Linux edge node server where it's accessible by all team members. Thus, the most recently updated SQL code and HPL/SQL procedures will be available to all members of your team. This is much better than having production code located across team member laptops…'nuf said!

☐ Can you please create a directory in HDFS specifically for me and my team for use with external tables? Something like **hdfs://hdpserver/data/prod/teams/<*schema*>** or whatever your standard is.

This directory and the subdirectories below it are where all of your Hadoop **external** database tables will be stored. Any time you create a table in Hadoop, a directory with that table name (for the most part) is created and your data is stored as one or more files under that directory. When you specify the `PURGE` option, dropping an external table causes the underlying file(s) and subdirectory(ies) to be deleted as well.

☐ I feel completely comfortable downloading and maintaining many of my department's dimension tables, but some of the fact tables are quite large. I'm hoping you can intercept the process involved in importing the fact

`tables and incorporate them into your process. Can we have a conversation about that?`

Since I'm not familiar with the process you go through to obtain data, let's assume that you have an outside vendor which provides your company with large delimited files on a timely basis. It may or may not be your job to load in these files into the legacy database. If it is, you can talk to the Hadoop Administrator directly and coordinate with him/her as to where to store these files, when/how often they should be loaded into the Hadoop database to minimize impact on the database, etc. If it's not your job, then you may want to coordinate with your Hadoop Administrator as well as your legacy database administrators. Together, y'all may find a simple way to load this data instead of having to drag it across the network directly from your legacy database using `sqoop`.

☐ What are the version numbers for the following?
- Linux (on the edge node server)
- Apache Hadoop
- Hive
- Impala
- HPL/SQL
- Hive ODBC Driver
- Impala ODBC Driver
- Hive JDBC Driver
- Impala JDBC Driver

It's important to know the version numbers of the software so that when you use the online documentation, you use the right version of the docs.

☐ Can you please install the Linux utility dos2unix on the Linux edge node server? Since our laptops are Windows-based, we may need to convert files using dos2unix.

Occasionally, when using files you've downloaded from the Internet or transferred from your Windows laptop to the Linux edge node server, loading data directly into the Hadoop database doesn't work as expected. Some issues can be overcome by running the utility `dos2unix` on the file(s) to convert Windows-style carriage returns/line feeds into Linux-style line feeds.

☐ Which Thrift Transport Mode should we be using? SASL? Binary? HTTP?

When setting up ODBC and other connections, you'll need to know which transport mode is expected by the Hadoop database. Generally, `SASL` is probably being used in your organization, but it's best to check with your Hadoop Administrator.

☐ Does the Hadoop Database use Secure Sockets Layer (SSL) for connections? When I go to set up an ODBC connection, there's an option asking whether I should enable SSL. Should I?

Your Hadoop Administrator will let you know if the database has been set up to make use of SSL and, if so, the SSL option in the Windows ODBC applet, as well as other connection strings, should be set to yes; otherwise, no.

☐ My team and I will be using the storage formats TEXTFILE, PARQUET and KUDU almost exclusively. Can you please indicate the SQL CREATE TABLE options required to use the KUDU storage format, if any? Can you recommend the number of partitions we should use with KUDU tables? Do we have to include the table property **kudu.master_addresses** in our SQL code? If so, can you include an example of this?

We talk about storage formats in detail below, but whereas TEXTFILE and PARQUET don't require any special options to work, depending on how your Hadoop Administrator has set up the database, KUDU **may** require additional options specified on the CREATE TABLE Statement. We discuss this more later in the book.

☐ In our legacy *<insert name of legacy database>* database, we have access to useful metadata such as table names, column names, data types, etc. within the database via ALL_TABLES, ALL_TAB_COLUMNS, INFORMATION_SCHEMA, etc. Can you create a view or views to mimic this from within the Hadoop database accessible from our new database schema? If not, can you give us read-only access to the underlying MetaStore database's metadata tables/views?

No doubt, you and your team make heavy use of the metadata views available in your legacy database. In Hadoop, strangely enough, the metadata isn't necessarily located in the Hadoop database itself, unlike other databases. For example, in Oracle, the views ALL_TABLES and ALL_TAB_COLUMNS allow you to peruse the metadata. Now, your Hadoop Administrator may be able create a view to the metadata accessible from within the Hadoop database itself. But, if the metadata is stored in an external database, such as MySQL, access is via the mysql command line utility. This isn't as drastic as it sounds because you can either create a table in Hadoop (via a process similar to that shown in *Chapter 1 – Quick Start Guide* for the US state name table), or you can query the external database directly via HPL/SQL in your procedures. We show both methods later in the book. Note that in Hive version 3 the sys database schema is available to use instead, but it's currently not accessible from Impala. We talk more about all this hullabaloo in *Chapter 33 – Accessing the Hive MetaStore*.

☐ Does the version of ImpalaSQL installed on the Hadoop database include the extensions to GROUP BY such as CUBE, ROLLUP, GROUPING SETS, etc.?

Depending on the version of ImpalaSQL installed, the extensions to the GROUP BY Clause may not be available. Knowing this up-front will allow you to plan what SQL code changes you'll need to make in order to work around this horrific loss. Boo-hoo!

☐ Is Apache Spark installed on the Linux edge node server? If so, what's the version number? As I would like to use Spark with Python, is pyspark available to use?

In case you want to use Spark with Python, you can do so via the command line utility pyspark. We don't discuss Spark in this book.

☐ My Team and I may create one or more user-defined functions (UDFs) for Impala. Can you create a directory in HDFS where we may place our Java .jar files? Also, can you update the PATH and CLASSPATH so that we have access to java and javac?

Although ImpalaSQL boasts oodles of yummy functions, occasionally you'll need to create one or more functions of your own to make you and your Team's SQL life easier. You create your functions in Java and then package them up into a Java .jar file. It's this *Java archive file* which is copied over to an appropriate directory in HDFS where it can be seen by Impala and then subsequently used in your ImpalaSQL code after creating a user-defined function pointing to that Java .jar file. We talk more about this in *Chapter 40 – Creating User-Defined Functions (UDFs) for ImpalaSQL*.

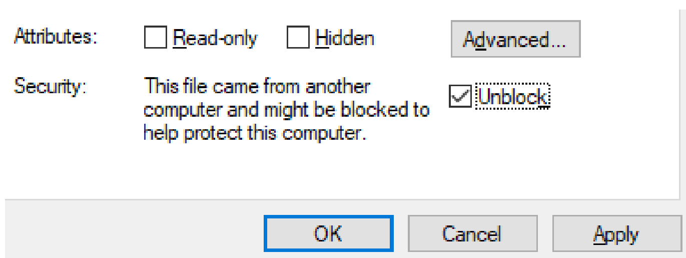
Chapter 3 – Recommended Windows Client Software

In this chapter, we discuss several useful applications for Windows you and your team should install (or be installed by your company's Desktop Services department, if that's necessary) on your company laptops. Note that if one or more of your team members will be using the web-based interface Hue exclusively, it may not be necessary to install these applications on their laptops, although I highly recommend the ODBC drivers be installed regardless.

First, though, you may want to create a folder on your corporate network to contain all of this software. This way, either your team members can install the software themselves from there, or Desktop Services can grab the software directly from that folder. Either way, you can control what software is available in that folder. For the discussion below, we assume you've created a folder on the corporate network named `\\corp\dept\software`. If you're doing this by yourself, maybe create `C:\TEMP\SOFTWARE` on your laptop.

In the text that follows, we assume that your Hadoop Administrator has responded to the Hadoop Administrator E-Mail. If not, you can still install the software, but you won't be able to do much except admire your **VAST DOMAIN OF SOFTWARE GOODNESS**. **MWA HA HA**!! Once you've received a response back from your Hadoop Administrator, you can then set up the software to access the Hadoop database and Linux edge node server.

Recall that, when downloading software from the *InterWebs*, Microsoft Windows may block the software. You can unblock the software by right-clicking over the filename, clicking the Properties menu item, checking Unblock and clicking the Apply button:



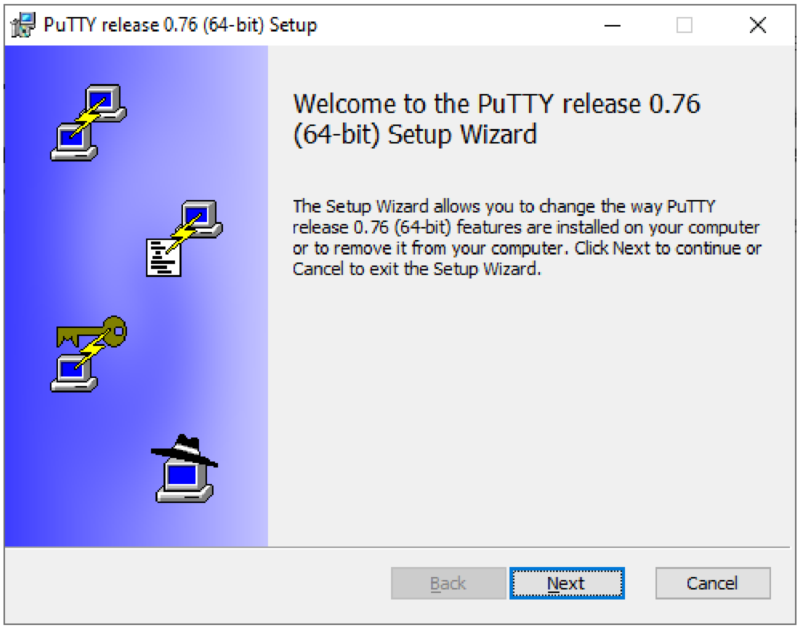
Before installing any piece of software on your company laptop, please run it through your company's virus protection software such as Windows Defender, Norton AntiVirus, McAfee Antivirus, etc. And, seriously, you should probably check with your Desktop Services department before installing any piece of software on your company laptop just in case a corporately-blessed, commercially-licensed, uninfected version of the software is already available in the company larder.

PuTTY

PuTTY, at its core, is used to log into a remote Linux server allowing you to issue commands from the command line. Since PuTTY is mainly used by programmers, not all users need to install it. The download and installation instruction follow.

Please create a folder labeled `PuTTY` under `\\corp\dept\software`.

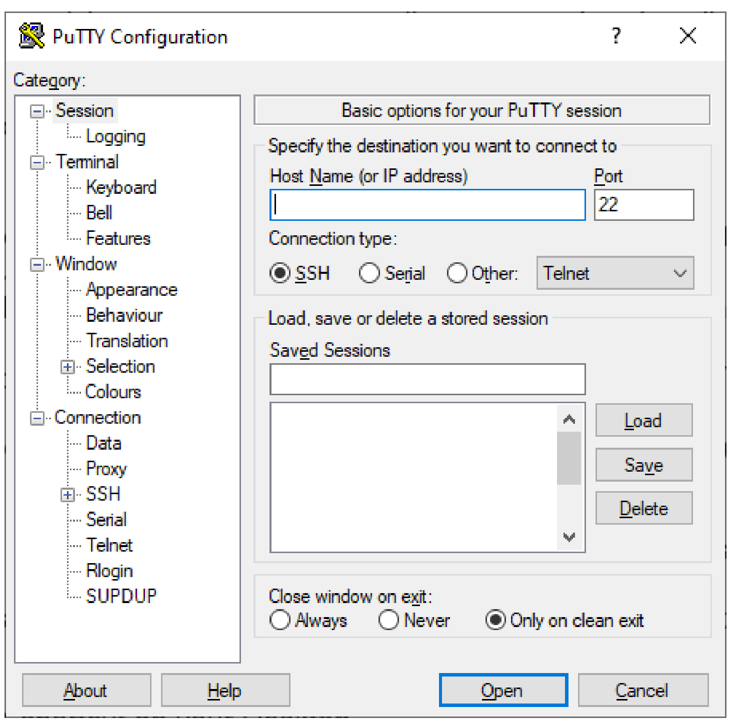
- ☐ Navigate your browser to `https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html`.
- ☐ Under the Packages files section, right click the link `putty-64bit-0.##-installer.msi` (where `##` is the latest version number), click on the Save link as… pop-up menu item and then save this file to the folder `\\corp\dept\software\`**`PuTTY`**.
- ☐ To install PuTTY, double-click `putty-64bit-0.##-installer.msi` and the Setup dialog will appear.



- ☐ Click the Next button.
- ☐ On the Destination Folder dialog box, specify the location where you want PuTTY to be installed.
- ☐ Click the Next button.
- ☐ On the Product Features dialog box, ensure all options are set to **Entire feature will be installed on local hard drive**.
- ☐ Click the Install button.
- ☐ Once PuTTY has finished installing the software, click the Finish button to close the installer.

At this point, a link to PuTTY should appear on your Windows Desktop. Let's set up the software to access your Linux edge node server.

- ☐ Double-click the PuTTY shortcut on your Desktop.
- ☐ The PuTTY Configuration dialog box should appear.



- ☐ In the **Host Name (or IP address)** input box, enter the edge node server's host name given to you by your Hadoop Administrator.
- ☐ Ensure the radio button to the left of the text **SSH** is selected.
- ☐ Ensure that the port number is set to 22 (or the port number indicated by your Hadoop administrator).
- ☐ In the Category tree at the left of the dialog box, click **Bell** under the **Terminal** branch and ensure that the radio button to the left of the text **None (bell disabled)** is selected. This will prevent an annoying **ding** each time you hit the Escape button in the `vi` editor (we talk more about that later on in the book).
- ☐ In the Category tree at the left of the dialog box, click the **Window** branch and update the **Lines of scrollback** input box to 20000. Since many Hadoop commands can cause *mega-mucho* amounts of lines to scroll past the window at breakneck speed, increasing this number allows you to use the scrollbar to go back to see what happened.
- ☐ In the Category tree at the left of the dialog box, click the Connection branch and update the **Seconds between keepalives (0 to turn off)** and update to 300. Occasionally, your session may close and PuTTY displays the aloof message **Connection reset by peer**. Updating to 300 seconds (5 minutes) may help prevent this. Note that if you continue to receive the snooty **Connection reset by peer** message, contact your Hadoop or Linux Administrator.
- ☐ In the Category tree at the left of the dialog box, click on the **Session** branch and enter in a friendly name of your session in the input box just below the text **Saved Sessions**.
- ☐ Click the Save button to save this session. Your session has now been saved and each time you run PuTTY, it will be available for you to use.
- ☐ Close PuTTY.

Now, to connect to the Linux edge node server from PuTTY, perform the following:

- ☐ Double-click the PuTTY shortcut on your Desktop.
- ☐ Double-click the friendly name of your saved session.
- ☐ The first time you start this session, PuTTY may display the **PuTTY Security Alert** dialog box indicating that the server's host key is not cached in the registry. Click Yes. From this point forward, you shouldn't see this message (unless a rebel like you adds another server to PuTTY! Naughty!).
- ☐ At this point, an ominous black screen pops up with the text **login as:**. Enter in your username, which is most likely the user name you use to log into your company laptop each morning (but, check with the responses to the Hadoop Administrator E-Mail).
- ☐ Hit the Enter key.
- ☐ Next, enter in the corresponding password.
- ☐ Hit the Enter key.

At this point, you are logged in to the Linux edge node server. If you're not familiar with Linux, this might seem pants-browningly scary! But don't worry, we discuss Linux later on in the book.

The strange text you see on the screen …

```
[smithbob@lnxserver ~]$
```

…indicates the Linux command prompt. It lovingly reminds you of your username (`smithbob`) as well as the hostname of the Linux server (`lnxserver`), just in case you've bonked your head and forgotten. After the dollar sign, you enter a Linux command and then hit the Enter key to execute it. We'll hold off on Linux commands until we get to the appropriate chapter. For now, type in the command `exit` and hit the Enter key. This will end your Linux session and PuTTY will close. Huzzah!

ODBC Drivers

Let's install both the Hive and Impala ODBC drivers. The instructions below assume you're using the Cloudera flavor of Hadoop, but the installation instructions should be about the same for other flavors. Note that if your Hadoop Administrator gave you a Corporate network location for the drivers, install those (or have those installed) instead.

Please create a folder labeled `ODBCDrivers_64bit` under `\\corp\dept\software`.

Let's download the Cloudera ODBC Driver for Apache Hive:

- ☐ Navigate your browser to `https://www.cloudera.com/downloads/connectors/hive/odbc/2-6-11.html`. Note that the webpage allows you to select a different version of the driver in the drop-down box under the text **Version:**. Ensure that you are downloading the appropriate version of the driver based on the responses from the Hadoop Administrator E-Mail.
- ☐ Select the appropriate operating system from the drop-down box labeled **SELECT AN OS**. For Microsoft Windows, select **Windows**.
- ☐ Select the appropriate version from the drop-down box labeled **SELECT A VERSION**. For Microsoft Windows, select **64-bit**. (Note that if you need the 32-bit driver, download those separately and place them in a folder named `ODBCDrivers_32bit` under `\\corp\dept\software`.)
- ☐ Click the **GET IT NOW!** button.
- ☐ Note that some sites require you to register, so you may need to go through a snooze-inducing registration process before you can download the drivers.
- ☐ Finally, download the Cloudera ODBC Driver for Apache Hive and save it in the `ODBCDrivers_64bit` folder. You may want to rename the file to reflect the version number of the driver.

Next, let's download the Cloudera ODBC Driver for Impala:

- ☐ Navigate your browser to `https://www.cloudera.com/downloads/connectors/impala/odbc/2-6-11.html`. Note that the webpage allows you to select a different version of the driver in the drop-down box under the text **Version:**. Ensure that you are downloading the appropriate version of the driver based on the responses from the Hadoop Administrator E-Mail.
- ☐ Select the appropriate operating system from the drop-down box labeled **SELECT AN OS**. For Microsoft Windows, select **Windows**.
- ☐ Select the appropriate version from the drop-down box labeled **SELECT A VERSION**. For Microsoft Windows, select **64-bit**. (Note that if you need the 32-bit driver, download those separately and place them in the folder `ODBCDrivers_32bit` under `\\corp\dept\software`.)
- ☐ Click the **GET IT NOW!** button.
- ☐ Finally, download the Cloudera ODBC Driver for Impala and save it in the `ODBCDrivers_64bit` folder. You may want to rename the file to reflect the version number of the driver.

Next, if you have Administrator rights on your laptop, you can install the ODBC drivers yourself. If not, you will have to ask your corporate Desktop Services to install them for you. Assuming you do have Administrator rights on your laptop, follow these instructions to install the ODBC drivers.

Let's first install the Cloudera ODBC Driver for Apache Hive:

- ☐ Navigate to `\\corp\dept\software\ODBCDrivers_64bit`.
- ☐ Double-click `ClouderaHiveODBC64_#.#.#.msi` to start the installer.
- ☐ On the **Welcome** dialog box, click Next.
- ☐ On the **End-User License Agreement** dialog box, ensure the checkbox to the left of the text **I accept the terms in the License Agreement.** is checked.
- ☐ Click Next.
- ☐ On the **Destination Folder** dialog box, assuming you want the driver installed in its default directory, click Next.
- ☐ On the **Ready to Install Cloudera ODBC Driver for Apache Hive** dialog box, click Install.
- ☐ The **Installing Cloudera ODBC Driver for Apache Hive** dialog box will appear for a while…be patient…grab a sandwich…
- ☐ On the **Completed** dialog box, click Finish to close the installer.

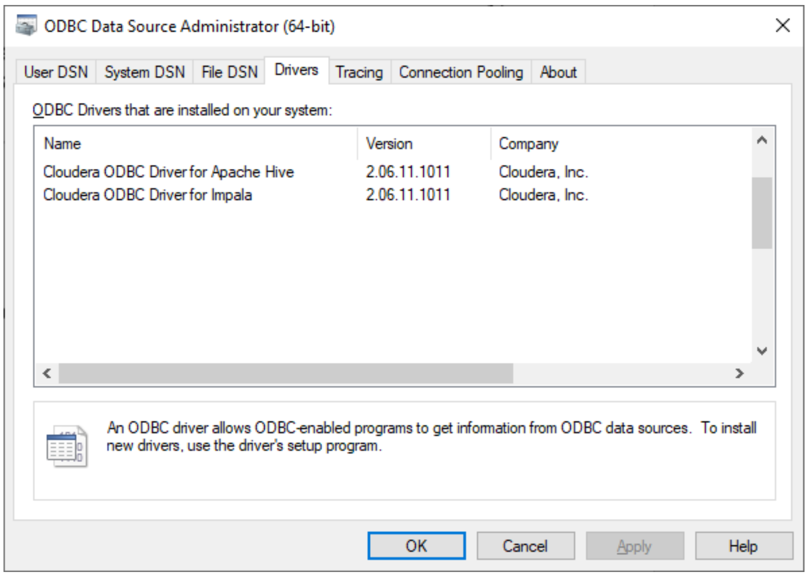
Next, let's install the Cloudera ODBC Driver for Impala driver:

- ☐ Navigate to `\\corp\dept\software\ODBCDrivers_64bit`.
- ☐ Double-click `ClouderaImpalaODBC64_#.#.#.msi` to start the installer.
- ☐ On the **Welcome** dialog box, click Next.
- ☐ On the **End-User License Agreement** dialog box, ensure the checkbox to the left of the text **I accept the terms in the License Agreement.** is checked.

- Click Next.
- On the **Destination Folder** dialog box, assuming you want the driver installed in its default directory, click Next.
- On the **Ready to Install Cloudera ODBC Driver for Apache Impala** dialog box, click Install.
- On the **Completed** dialog box, click Finish to close the installer.

Since both 64-bit ODBC drivers are now installed, let's open the Windows ODBC Data Sources (64-bit) applet to check they are actually available to use:

- Click the Start button and begin typing the letters ODBC. You should see several suggestions, one of which is **ODBC Data Sources (64-bit)**. Click on the link to start the applet.
- When the applet appears, click on the Drivers tab and you should see something similar to the following:



Since both drivers appear on the Drivers tab, you should now be able to use ODBC to connect to both Hive and Impala. Note that if you installed the 32-bit drivers as well, they will appear in the ODBC Data Source Administrator (**32-bit**) applet. You can tell the difference between the 64-bit and 32-bit drivers by the name of the driver files indicated under the File column on the Drivers tab (in the image above, you need to scroll to the right a tad bit). When perusing the ODBC Data Source Administrator (**64-bit**) applet, the files end in **64**. When perusing ODBC Data Source Administrator (**32-bit**) applet, the files end in **32**. Good system, huh?

JDBC Drivers

Let's install both the Hive and Impala JDBC drivers. The instructions below assume you're using the Cloudera flavor of Hadoop, but the installation instructions are about the same for other flavors. Note that if your Hadoop Administrator gave you a corporate network location for the drivers, use those instead.

Note that you will pull **specific versions** of the JDBC drivers, one for Hive and one for Impala. This is because additional Java `.jar` files are included that are not included in later versions of the files. This makes me sad! If this is an issue, please contact your Hadoop Administrator (not about me being sad…you know what I mean…).

Please create a folder labeled `JDBCDrivers` under `\\corp\dept\software`.

Let's download the Hive JDBC Connector:

- Navigate your browser to `https://www.cloudera.com/downloads/connectors/hive/jdbc/2-6-15.html`. Note that the webpage allows you to select a different version of the driver in the drop-down box under the text **Version:**, ***but leave the selection set to 2.6.15***.

- ☐ Depending on the version of the driver you want to download, you may not be asked to select additional operating system options. If asked, select the same options used with the ODBC drivers above.
- ☐ Click the **GET IT NOW!** button.
- ☐ Finally, save the Hive JDBC Connector (named `hive_jdbc_2.6.15.108.zip`) in the `JDBCDrivers` folder.

Next, let's download the Impala JDBC Connector:

- ☐ Navigate your browser to `https://www.cloudera.com/downloads/connectors/impala/jdbc/2-5-45.html`. Note that the webpage allows you to select a different version of the driver in the drop-down box under the text **Version:**, ***but leave the selection set to 2.5.45***! This particular download contains additional JAR files needed to allow the ImpalaSQL JDBC driver to function properly. The newer ImpalaSQL JDBC drivers do not contain these additional JAR files!
- ☐ Depending on the version of the driver you want to download, you may not be asked to select additional operating system options. If asked, however, select the same options used with the ODBC drivers above.
- ☐ Click the **GET IT NOW!** button.
- ☐ Finally, download the Cloudera JDBC Driver for Impala (named `impala_jdbc_2.5.45.1065.zip`) and save it in the `JDBCDrivers` folder.

Now that the JDBC drivers have been downloaded, let's unpack them. First, the Hive JDBC Connector:

- ☐ On your laptop, create a folder `C:\TEMP\JDBCDrivers`.
- ☐ Double-click the file `hive_jdbc_2.6.15.108.zip` to start WinZip (or other decompression utility).
- ☐ When WinZip starts, double-click the folder `hive_jdbc_2.6.15.108`.
- ☐ Drag the folder named `ClouderaHiveJDBC41-2.6.15.1018` from WinZip to the folder `C:\TEMP\JDBCDrivers`. WinZip should decompress the folder and place it and its contents in the `JDBCDrivers` folder.
- ☐ Navigate to the folder `C:\TEMP\JDBCDrivers\ClouderaHiveJDBC41-2.6.15.1018`. You should see the single Java `.jar` file named `HiveJDBC41.jar`.

Next, the Impala JDBC Connector:

- ☐ Double-click the file `impala_jdbc_2.5.45.1065.zip` to start WinZip (or other decompression utility).
- ☐ When WinZip starts, double-click the folder `ClouderaImpalaJDBC_2.5.45.1065`.
- ☐ Double-click the WinZip file `ClouderaImpalaJDBC41_2.5.45.zip`.
- ☐ When WinZip starts (again), you'll see the folder `ClouderaImpalaJDBC41_2.5.45`. Drag this folder from WinZip to the folder `C:\TEMP\JDBCDrivers`. WinZip should decompress the folder and place it and its contents in the `JDBCDrivers` folder.
- ☐ Navigate to the folder `C:\TEMP\JDBCDrivers\ClouderaImpalaJDBC41_2.5.45`. **You should see several JAR files including `ImpalaJDBC41.jar`**.

Java Runtime Environment (JRE)

Some of the SQL clients described below require the Java Runtime Environment (JRE) to work properly. Note that some SQL clients may, in fact, include the JRE as part of the download. Before we describe how to download and install the JRE, let's see if you already have a version of it installed:

- ☐ Open up Windows Explorer and navigate to `C:\Program Files\Java`. Under that folder, if you see one or more folders prepended with the letters `jre` or `jdk`, you're probably good to go. The folder prepended with the letters `jre` contains the Java Runtime Environment (JRE) whereas the folder prepended with the letters `jdk` contains the full Java Development Kit (JDK). The JDK should contain similar executables to the JRE.
- ☐ Navigate down to the `bin` folder under the `jre` folder, if you have it; otherwise, use the `jdk` folder instead. Copy the full directory location from the address bar at the top of Windows Explorer.

- ☐ Open up the Windows Command Prompt by clicking the Start button and begin typing the text `command`. Click the Command Prompt icon when it appears. When the Command Prompt dialog box appears, type in the letters `cd` (which stand for *change directory*…hmmm!…they have the same thing in the Linux operating system…verrrry interesting!) followed by the location you copied in the previous step, and hit the Enter key:

```
C:\Users\smithbob> cd C:\Program Files (x86)\Java\jre7\bin
```

You are now located in the `bin` subdirectory. Next, enter in the following command and hit the Enter key:

```
C:\Program Files (x86)\Java\jre7\bin> java -version
```

For me, the output I see is the following:

```
java version "1.7.0_65"
Java(TM) SE Runtime Environment (build 1.7.0_65-b19)
Java HotSpot(TM) Client VM (build 24.65-b04, mixed mode, sharing)
```

This indicates that you already have the JRE, version 1.7.0_65, installed on your laptop. This may seem like a big win worthy of an expensive corporate-paid-for lunch, but some of the SQL clients we describe below may require a later version of the JRE. Note that when the version number is reported for both the JRE and the JDK, the second period-delimited number is reported; 7, in this case, indicates it's version 7 of the JRE. Close the Command Prompt by typing the text `exit` and violently smashing the Enter key.

If you have neither the JRE nor JDK installed on your laptop, you will need to install it. **You probably should check with your Desktop Services department first to see if they have a commercial version they would prefer you to install (or have them install).** In any case, the instructions for downloading and installing the non-commercial version of the Java Runtime Environment (JRE) appear below.

Create a folder named `java_runtime_environment` under `\\corp\dept\software`.

Let's first download the latest JRE:

- ☐ Navigate your browser to `https://www.java.com/en/download/manual.jsp`.
- ☐ Click on the link labeled **Windows Offline (64-bit)**.
- ☐ The download for the JRE should start immediately. The file being downloaded will be named something like `jre-8u###-windows-x64.exe`. Take note that the 8 indicates JRE version 8.
- ☐ Once the download completes, move this file to the `java_runtime_environment` folder.

Next, let's install the JRE:

- ☐ Double-click `jre-8u###-windows-x64.exe` to start the JRE installer.
- ☐ When the **Java Setup – Welcome** dialog box appears, ensure the checkbox to the left of the text **Change destination folder** is checked.
- ☐ Click the Install button.
- ☐ On the **Destination Folder** dialog box, take note of the folder location where the JRE will be installed. This may be used while setting up the SQL client software described below.
- ☐ Click the Next button to install the JRE. This may take a while… get yourself a nice baked good…
- ☐ The **Out-of-Date Java versions Detected** dialog box may appear. To prevent some of your Java-based software from crapping out, click the Not Now button to preserve these older versions of Java.
- ☐ When the **Java Setup – Complete** dialog box appears, click the Next button.
- ☐ (If that weren't enough, let's go for one more dialog box, shall we?) When the **Java Setup – Complete** dialog box appears (again), click the Close button.

Now, the installation above should have placed the latest JRE folder in your `PATH` environment variable (a very important Microsoft Windows text string containing software locations…hmmm!…they have the same thing in the Linux operating system…verrrry interesting!). Let's double-check this:

- ☐ Open up the Windows Command Prompt by clicking the Start button and begin typing in the text `command`. Click the Command Prompt icon when it appears. Enter in the following command and hit the Enter key:

```
C:\Users\smithbob> java -version
```

  For me, the output I see is the following:

```
java version "1.8.0_301"
Java(TM) SE Runtime Environment (build 1.8.0_301-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.301-b09, mixed mode)
```

- ☐ At this point, you're good to go! Type `exit` and hit the Enter key to close the Windows Command Prompt.

ODBC Data Source Names (DSNs)

Occasionally, you may need to make use of ODBC Data Source Names (DSNs) in Windows applications such as Microsoft Excel, Microsoft Access, PowerBI, Tableau, etc. Since the ODBC drivers have been installed, let's create one DSN for Hive and one DSN for Impala.

Below, we describe the set up for the 64-bit drivers using the ODBC Data Sources (**64-bit**) applet. If you want corresponding DSNs that make use of the 32-bit drivers, use the ODBC Data Sources (**32-bit**) applet instead.

First, let's create a DSN for Hive:

- ☐ Start the ODBC Data Sources (64-bit) applet.
- ☐ Once the **ODBC Data Sources Administrator (64-bit)** applet dialog box appears, click on the **User DSN** tab.
- ☐ Click the Add… button to create a new DSN.
- ☐ Once the **Create New Data Source** dialog box appears, click on the entry for **Cloudera ODBC Driver for Apache Hive** to highlight it.
- ☐ Click Finish.
- ☐ The very rectangular **Cloudera ODBC Driver for Apache Hive DSN Setup** dialog box will appear. Fill in the input boxes like this:
  - Data Source Name: `ODBC_HIVE_`*`hdpserver`* (replace *`hdpserver`* with your Hadoop server name throughout)
  - Description: `Apache Hive ODBC Connection to` *`hdpserver`*
  - Hive Server Type: Hive Server 2
  - Service Discovery Mode: No Service Discovery
  - Host(s): *`hdpserver`*
  - Port: 10000
  - Database: *<enter the name of your default schema, like prod_schema>*
  - Mechanism: User Name and Password
  - User Name: *<enter in your username>*
  - Password: *<enter in your password>*
  - Thrift Transport: *<enter the name of the transport provided by your Hadoop Administrator>*
  - Click on Advanced Options and ensure the checkbox to the left of the text **Use Native Query** is checked.
- ☐ Click the Test button to test out the connection. If all goes well, you should be greeted with a SUCCESS! message. If not, check your entries as well as the responses provided by your Hadoop Administrator in the Hadoop Administrator E-Mail.

Next, let's create a DSN for Impala:

- ☐ Start the ODBC Data Sources (64-bit) applet.
- ☐ Once the **ODBC Data Sources Administrator (64-bit)** applet dialog box appears, click on the **User DSN** tab.
- ☐ Click the Add… button to create a new DSN.

- ☐ Once the **Create New Data Source** dialog box appears, click on the entry for **Cloudera ODBC Driver for Impala** to highlight it.
- ☐ Click Finish.
- ☐ The **Cloudera ODBC Driver for Impala DSN Setup** dialog box will appear. Fill in the input boxes like this:
  - Data Source Name: `ODBC_IMPALA_`*`hdpserver`* (replace *`hdpserver`* with your Hadoop server name)
  - Description: `Impala ODBC Connection to `*`hdpserver`*
  - Host(s): *`hdpserver`*
  - Port: 21050
  - Database: *<enter the name of your default schema>*
  - Mechanism: User Name and Password
  - User Name: *<enter in your username>*
  - Password: *<enter in your password>*
  - Ensure the checkbox to the left of the text **Save Password (encrypted)** is checked
  - Transport Buffer Size: 50000
  - Transport Mode: *<enter the name of the transport provided by your Hadoop Administrator>*
  - Click on Advanced Options and ensure the checkbox to the left of the text **Use Native Query** is checked.
- ☐ Click the Test button to test out the connection. If all goes well, you should be greeting with a SUCCESS! message. If not, check your entries as well as the responses provided by your Hadoop Administrator in the Hadoop Administrator E-Mail.

Note that you may also want to set up DSNs using the generic account's username/password as well.

Toad Data Point

Toad Data Point is a multi-platform database query, data preparation and reporting tool. Unlike other software in this section, it's not free…you have to pay through the nose for it. My team and I use Toad Data Point and it's a very professional SQL query client, but – in my opinion – it's a bit overkill with features you may never actually use. With that said, you and your team should give Toad Data Point a look-see.

Create a folder named `ToadDataPoint` under `\\corp\dept\software`.

Let's download the 30-day trial of Toad Data Point.

- ☐ Navigate your browser to `https://support.quest.com/toad-data-point/4.1/download-new-releases`.
- ☐ Select the latest version from the filter drop-down box and you'll be sent to the appropriate webpage.
- ☐ Under the section labeled Software, click the link for **Toad Data Point #.# Base & Professional Installer (for 32-bit or 64-bit)**.
- ☐ Click the **Add to Downloads** button.
- ☐ On the **Sign In** webpage, you'll need to go through the hassle of signing up…I know, I know…such a pain!
- ☐ Once signed in, complete the download and store the installer in the `ToadDataPoint` folder.

Let's install Toad Data Point.

- ☐ Navigate to the `ToadDataPoint` folder and double-click the installer (named something like this `ToadDataPoint_pro_#.#.#.exe`)
- ☐ On the **Choose Installer** dialog box, select **Install 64-bit Toad Data Point**.
- ☐ Click Next.
- ☐ On the **Welcome to Toad** dialog box, click Next.
- ☐ On the **End-User License Agreement** dialog box, laugh hysterically, click the checkbox and then click Next.
- ☐ On the **Destination Folder** dialog box, click Next.
- ☐ On the **Install Type** dialog box, ensure the radio button to the left of the text **Typical installation** is selected.
- ☐ Click Next.

- ☐ On the **Register File Extensions** dialog box, leave the Toad File Extensions checked, but decide whether you want `.sql` files to open in Toad when double-clicked.
- ☐ Click Next.
- ☐ On the **Additional Properties** dialog box, ensure that the radio button to the left of the text **Allow saving passwords** is selected. Note that you should be cautious when using this option and adhere to Corporate policy!
- ☐ Click Next.
- ☐ On the **Ready to Install Toad Data Point #.# (64-bit)**, click the Install button.
- ☐ The installation may take a while, so why not take the time to tear a pheasant with some loved-ones?
- ☐ On the **Toad Data Point #.# (64-bit) successfully installed** dialog box, ensure the checkbox to the left of the text **Show release notes** is unchecked (release notes…booooooooring!!) and the radio button to the left of the text **Do Nothing** is selected.
- ☐ Click Next.

Let's set up Toad Data Point to connect to the Hadoop database. Note that since we set up two ODBC connections above, we'll make use of those during the set up. First, let's set up Hive.

- ☐ Start Toad Data Point by double-clicking the shortcut labeled Toad Data Point #.# on your Desktop.
- ☐ On the left side of the GUI interface is the Navigation Manager. Click the icon labeled **Create a new connection**.
- ☐ When the **Create New Connection** dialog box appears, enter the word `odbc` in the **Type to filter list** input box.
- ☐ When the text ODBC Generic appears, click it.
- ☐ Two tabs will appear, one labeled General and the other Advanced. Click on the **General** tab and fill in the dialog like this (replacing entries at will):
    - Ensure the checkbox to the left of the text **Use data source** name is checked.
    - Select the ODBC connection `ODBC_HIVE_`*`hdpserver`* from the Data source name drop-down box.
    - User: *<enter your username>*
    - Password: *<enter your password>*
    - Database: *prod_schema*
- ☐ Click the Save button.

Next, let's set up Impala.

- ☐ On the left side of the GUI interface is the Navigation Manager. Click the icon labeled **Create a new connection**.
- ☐ When the **Create New Connection** dialog box appears, enter the word `odbc` in the **Type to filter list** input box.
- ☐ When the text ODBC Generic appears, click it.
- ☐ Two tabs will appear, one labeled General and the other Advanced. Click on the **General** tab and fill in the dialog like this (replacing entries at will):
    - Ensure the checkbox to the left of the text **Use data source** name is checked.
    - Select the ODBC connection `ODBC_IMPALA_`*`hdpserver`* from the Data source name drop-down box.
    - User: *<enter your username>*
    - Password: *<enter your password>*
    - Database: *prod_schema*
- ☐ Click the Save button.

At this point, you should check both connections by right-clicking the connection names, and clicking the Connect menu item. If you connect, then you're good to go! If not, please check the connection information.

Note that you do not need to create DSNs to connect Toad Data Point to Hive and Impala. In the instructions above, instead of choosing **ODBC Generic**, select either **Apache Hive** or **Cloudera Impala** from the drop-down box.

For Apache Hive, fill in the dialog like this:

- ☐ On the Server tab, fill the entries like this:

  - Host: *hdpserver*
  - Port: 10000
  - Schema: *prod_schema*
  - Server Type: HiveServer2
  - Use SSL: *<check the box if your Hadoop Administrator indicated that the Hadoop database is set up to use SSL; otherwise, leave it unchecked>*
  - HTTP Mode: *<fill this in based on the responses to the Hadoop Administrator E-Mail>*
- ☐ On the Authentication tab, fill the entries like this:
  - Authentication: *<select the appropriate entry from the drop-down; most likely **Username and password**>*
  - Username: *<enter your username>*
  - Password: *<enter your password>*
- ☐ Click the Save button to save this connection.

For Cloudera Impala,

- ☐ On the General tab, fill the entries like this:
  - Uncheck *Use data source name*
  - Driver name: Cloudera ODBC Driver for Impala
  - User: *<enter your username>*
  - Password: *<enter your password>*
  - Database: *prod_schema*
  - Connection String: *<modify the following string and place it in the input box to the right of the text ConnectionString:>*

```
Driver=Cloudera ODBC Driver for Impala; Host=hdpserver; Port=21050; AuthMech=3;
KrbRealm=<your-KrbRealm>; KrbHostFQDN=<your-KrbHostFQDN>; KrbServiceName=impala;
UseNativeQuery=0;
```

Note that you should remove the Kerberos-related entries if you company does not use Kerberos. The `AuthMech` entry allows for the following values:

AuthMech	Description
0	No authentication.
1	Kerberos authentication.
2	User name authentication.
3	User name and password authentication.
4	User name and password authentication with SSL enabled.

- ☐ Click the Save button to save this connection.

DBeaver Universal Database Manager

DBeaver Universal Database Manager is a free SQL integrated development environment (IDE) which allows for connections to Hive and Impala via the JDBC drivers installed earlier as well as connections to Oracle, Teradata, Access, MongoDB, etc. via their own JDBC drivers (which you'll have to download and install separately). Let's download, install and set up DBeaver SQL to access your Hadoop database.

Note that DBeaver requires the Java Runtime Environment (JRE) to run.

Please create a folder labeled `DBeaver` under `\\corp\dept\software`.

To download the DBeaver Universal Database Manager:

- ☐ Navigate your browser to `https://dbeaver.io/`.
- ☐ Click the Download button.

- ☐ On the Download webpage, under the Community Edition column, download the Windows 64 bit (installer) and save it to the `DBeaver` folder.

Let's install DBeaver:

- ☐ Navigate to `\\corp\dept\software\DBeaver` and double-click on the installer `dbeaver-ce-#.#.#-x86_64-setup.exe`.
- ☐ On the **Installer Language** dialog box, select your language and click OK.
- ☐ On the **Welcome to DBeaver Community Setup** dialog box, click Next.
- ☐ On the **License Agreement** dialog box, click I Agree.
- ☐ On the **Choose Users** dialog box, select **For me (smithbob)** and click Next.
- ☐ On the Choose Components dialog box, ensure the checkboxes to the left of the text **DBeaver Community** and the text **Include Java** are checked.
- ☐ Click Next.
- ☐ On the **Choose Install Location** dialog box, click Next.
- ☐ On the **Choose Start Menu Folder** dialog box, click Install.
- ☐ The installation will begin shortly…you may wanna order a pizza…
- ☐ When the **Completing DBeaver Community Setup** dialog box appears, ensure the checkbox to the left of the text **Create Desktop Shortcut** is checked.
- ☐ Click Finish.

Next, let's set up DBeaver to connect to the Hadoop database via Impala.

- ☐ Start DBeaver by double-clicking on the shortcut on the Desktop.
- ☐ On the **Create sample database** dialog box, click No.
- ☐ On the **Select your database** dialog box, click the All link (on the left side).
- ☐ In the input box containing the text **Type part of database/driver name to filter**, type `impala`. An icon for Cloudera Impala should appear. Click on that icon and click Next.
- ☐ On the **Generic JDBC Connection Settings** dialog box, fill in the entries like this:
  - Host: *hdpserver*
  - Port: 21050
  - Database/Schema: *prod_schema*
  - Username: *<enter your username>*
  - Password: *<enter your password>*
  - Ensure the checkbox to the left of the text **Save password locally** is checked.
- ☐ Click the Edit Driver Settings button.
- ☐ When Edit Driver 'Cloudera Impala' appears, click the **Libraries** tab.
- ☐ Click the Add Folder button.
- ☐ Navigate to the folder on disk where you placed the JAR files related to the Cloudera JDBC Connector for Impala. Click Select Folder.
- ☐ To the right of the drop-down box after the text Driver Class, click the Find Class button.
- ☐ In the drop-down box, select `com.cloudera.impala.jdbc41.Driver`.
- ☐ Click on the Driver properties tab:
  - Right-click and click the **Add new property** menu item.
  - In the Property Name input box, type in `AuthMech` and click OK.
  - Click on the value column to the right of `AuthMech` and enter in the number `3`, if appropriate.
  - Click OK.
- ☐ Click the Test Connection… button. If all goes well, a Connected dialog box will appear. Click OK to dismiss this box.
- ☐ Click OK to close the **Connection settings** dialog box.
- ☐ On the Database Navigator tab, you should see an entry for `prod_schema`. Double-click it and DBeaver should connect to the Hadoop database via Impala.

Next, let's set up DBeaver to connect to the Hadoop database via Hive.

- ☐ Start DBeaver by double-clicking on the shortcut on the Desktop.
- ☐ Click the New Database Connection icon on the upper left of the IDE just above the Database Navigator tab.

- ☐ On the Select your database dialog box, click the All link (on the left side).
- ☐ In the input box containing the text **Type part of database/driver name to filter**, type `hive`. An icon for Apache Hive should appear. Click on the icon and click Next.
- ☐ On the Generic JDBC Connection Settings dialog box, fill in the entries like this:
  - Host: *hdpserver*
  - Port: 10000
  - Database/Schema: *prod_schema*
  - Username: *<enter your username>*
  - Password: *<enter your password>*
  - Ensure the checkbox to the left of the text **Save password locally** is checked.
- ☐ Click the Edit Driver Settings button.
- ☐ When Edit Driver 'Apache Hive' appears, click the **Libraries** tab.
- ☐ If an entry appears, highlight it and click Delete.
- ☐ Click the Add File button.
- ☐ Navigate to the folder on disk where you placed the JAR files related to the Cloudera JDBC Connector for Impala. Highlight all of the JAR files **except** for `ImpalaJDBC41.jar`. Click Open.
- ☐ Click Add File button once again.
- ☐ Navigate to the folder on disk where you placed the `HiveJDBC41.jar` file. Highlight it and click Open.
- ☐ To the right of the drop-down box after the text Driver Class, click the Find Class button.
- ☐ In the drop-down box, select `com.cloudera.hive.jdbc41.HS2Driver`.
- ☐ Click on the Driver properties tab:
  - Right-click and click the **Add new property** menu item.
  - In the Property Name input box, type in `AuthMech` and click OK.
  - Click on the value column to the right of `AuthMech` and enter in the number `3`, if appropriate.
  - Click OK.
- ☐ Click the Test Connection… button. If all goes well, a Connected dialog box will appear. Click OK to dismiss this box.
- ☐ Click OK to close the **Connection settings** dialog box.
- ☐ On the Database Navigator tab, you should see an entry for prod_schema. Double-click on it and DBeaver should connect to the Hadoop database via Apache Hive.

Note that you will see the same entry twice for `prod_schema`. You can right-click over each entry, click the Rename menu item and rename it to whatever you want. Don't be naughty, you dirty little ferret!!

SQuirreL SQL

Almost eerily similar to DBeaver, SQuirreL SQL is a free SQL integrated development environment (IDE) which allows for connections to Hive and Impala via the JDBC drivers installed earlier as well as connections to Oracle, Teradata, Access, MongoDB, etc. via their own JDBC drivers (which you will have to download and install separately). Let's install and set up SQuirreL SQL to access your Hadoop database.

Note that SQuirreL SQL requires the Java Runtime Environment (JRE) to run!

Please create a folder labeled `SQuirreL` under `\\corp\dept\software`.

To download the SQuirreL SQL client:

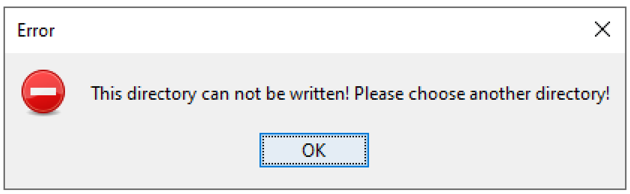
- ☐ Navigate your browser to `http://squirrel-sql.sourceforge.net/`.
- ☐ Click on the **Download Squirrel SQL Client** link.
- ☐ In the **Download and Installation** section of the webpage, click on the link labeled **Install jar of SQuirreL 4.2.0 for Windows/Linux/others**.
- ☐ At this point, you should be transferred to SourceForge.net and the download will begin.
- ☐ Note that if you receive a browser warning asking you if you want to keep the file, click the Keep button.
- ☐ Move the downloaded file `squirrel-sql-#.#.#-standard.jar` to the `SQuirreL` folder.

Next, let's install the software.

- Double-click the file `squirrel-sql-#.#.#-standard.jar`. Alternatively, you can start the installer from the Windows Command Prompt by running the following command:

```
java -jar squirrel-sql-#.#.#-standard.jar
```

- When the **Installation of SQuirreL SQL Client** dialog box appears, click the Next button.
- On the **Please read the following information** dialog box, take note of the SQuirreL SQL Client Version as well as the JRE Minimum Version. **If your installed version of JRE does meet these requirements, then it's a good idea to download and install the latest JRE (see the instructions above).**
- Click Next.
- On the **Select the installation path** dialog box, take note of the installation directory. Click the Next button.
- If you receive the following dialog box, you will have to select another directory. I usually create folders under `C:\TEMP`, but you can do your own *thang*! Click OK to dismiss the dialog box. I opted for `C:\TEMP\squirrel-sql-#.#.#` as the folder name.



- Click Next.
- If a dialog box appears indicating that the subdirectory will be created, click OK.
- On the **Select the packs you want installed** dialog box, select all Optional Plugins.
- Click Next.
- The installation will begin immediately. When complete, click the Next button.
- On the **Setup Shortcuts** dialog box, ensure that the checkbox to the left of the text **Create additional shortcuts on the desktop** is checked.
- Click Next.
- Finally, click Done to close the installer.

Next, let's setup the SQuirreL SQL Client to connect to your Hadoop database using the Impala JDBC Connector.

- Start SQuirreL by double-clicking the shortcut on your Windows Desktop.
- On the left side, click the **Drivers** tab. Click on the plus-sign to create a new driver.
- Within the Driver section, enter a name for your driver in the input box to the right of the text Name. For example, `Cloudera Impala JDBC Driver`.
- In the **Example URL** input box, place the following text (modify the `hdpserver` as well as `KrbRealm` and `KrbHostFQDN` for your Hadoop server, if necessary):

```
jdbc:impala://hdpserver:21050;AuthMech=3;KrbRealm=REALM.COMPANY.COM;KrbHostFQDN=hdpse
rver;KrbServiceName=impala;UseNativeQuery=1
```

- Place the same text as above in the **Website URL** input box.
- Click the **Extra Class Path** tab.
- Click the Add button.
- Navigate to where you stored the Impala JDBC Drivers (something like `C:\TEMP\JDBCDrivers\ClouderaImpalaJDBC41_2.5.45`) and **highlight all of the JAR files under this folder**.
- Click the Open button and the highlighted JAR files will be displayed in the **Extra Class Path** tab.
- Highlight the `ImpalaJDBC41.jar` file and click on the List Drivers button. At the very bottom of the dialog, the drop-down for Class Name should be filled in with `com.cloudera.impala.jdbc41.Driver`. If not, click on the drop-down arrow and select it.
- Click OK.

At this point, SQuirreL knows that the new driver exists, but in order to actually use it, we need to set up an alias which makes use of this new driver to the Hadoop database via Impala. Let's set up an alias to connect to the Hadoop database via Impala.

- ☐ In the **Drivers** tab, click the new driver **Cloudera Impala JDBC Driver**. This will be picked up by the alias automatically for the instructions below.
- ☐ Click the **Aliases** tab on the left side of SQuirreL.
- ☐ Click the plus-sign to create a new alias.
- ☐ When the **Add Alias** dialog box appears, fill in the input box to the right of the text Name with **ImpalaSQL on *hdpserver***.
- ☐ The Driver drop-down box should say **Cloudera Impala JDBC Driver**. If not, select it from the drop-down box.
- ☐ The URL should be a copy of the **Example URL** from the Drivers tab and should appear automatically.
- ☐ Fill in your username in the input box to the right of the text **User Name:**. Note that this may be your Windows log in username, or something else. Please check the Hadoop Administrator E-Mail responses.
- ☐ Fill in your password in the input box to the right of the text **Password:**.
- ☐ Ensure the checkbox to the left of the text **Save password encrypted** is checked.
- ☐ Click the Test button to check if you can connect to the Hadoop database. If so, excellent! If not, please check all of the information entered in both the alias as well as the driver. If you still cannot connect to the database, please work with your Hadoop Administrator.
- ☐ Click the OK button.

Now let's setup the SQuirreL SQL Client to connect to your Hadoop database using the Hive JDBC Connector.

- ☐ Start SQuirreL by double-clicking the shortcut on your Windows Desktop.
- ☐ On the left side, click the **Drivers** tab. Click on the plus-sign to create a new driver.
- ☐ Within the Driver section, enter a name for your driver in the input box to the right of the text Name. For example, `Cloudera Hive JDBC Driver`.
- ☐ In the **Example URL** input box, place the following text (change `hdpserver` for your Hadoop Hive connection indicated in response to the Hadoop Administrator E-Mail):

```
jdbc:hive2://hdpserver:10000/default;authMech=3;
```

- ☐ Place the following text in the **Website URL** input box:

```
https://www.cloudera.com/downloads/connectors/hive/jdbc/2-6-15.html
```

- ☐ Click the **Extra Class Path** tab.
- ☐ Click the Add button.
- ☐ Navigate to where you stored the Hive JDBC Drivers (something like `C:\TEMP\JDBCDrivers\ClouderaHiveJDBC41-2.6.15.1018`) and highlight the single JAR file `HiveJDBC41.jar` under this folder.
- ☐ Click the Open button and the highlighted JAR files will be displayed in the Extra Class Path tab.
- ☐ Click the Add button once again.
- ☐ Navigate to the folder where you stored the Impala JDBC Drivers (something like `C:\TEMP\JDBCDrivers\ClouderaImpalaJDBC41_2.5.45`) and highlight all of the JAR files under this folder **excluding the JAR file `ImpalaJDBC41.jar`**.
- ☐ Highlight the `HiveJDBC41.jar` file and click on the List Drivers button. At the very bottom of the dialog, select the driver named `com.cloudera.hive.jdbc41.HS2Driver` from the drop-down box for Class Name.
- ☐ Click OK.

At this point, SQuirreL knows that the new driver exists, but in order to actually use it, we need to set up an alias which makes use of this new driver to connect to the Hadoop database via Hive.

- ☐ In the Drivers tab, click the new driver **Cloudera Hive JDBC Driver**. This will be picked up by the alias automatically for the instructions below.
- ☐ Click the **Aliases** tab on the left side of SQuirreL.

- ☐ Click the plus-sign to create a new alias.
- ☐ When the Add Alias dialog box appears, fill in the input box to the right of the text Name with **HiveQL on *hdpserver***.
- ☐ The Driver drop-down box should say **Cloudera Hive JDBC Driver**. If not, select it from the drop-down box.
- ☐ The URL should be a copy of the **Example URL** from the Drivers tab.
- ☐ Fill in your username in the input box to the right of the text **User Name:**. Note that this may be your Windows log in username, or something else. Please check the Hadoop Administrator E-Mail responses.
- ☐ Fill in your password in the input box to the right of the text **Password:**.
- ☐ Ensure the checkbox to the left of the text **Save password encrypted** is checked.
- ☐ Click the Test button to check if you can connect to the Hadoop database. If so, excellent! If not, please check all of the information entered in both the alias as well as the driver. If you still cannot connect to the database, please work with your Hadoop Administrator.
- ☐ Click the OK button.

Next, let's try to connect to the Hadoop database.

- ☐ Start SQuirreL by double-clicking the shortcut on your Windows Desktop.
- ☐ Click the Aliases tab to display your aliases.
- ☐ Right-click over **ImpalaSQL on *hdpserver*** and click the Connect… menu item.
- ☐ When the **Connect to: ImpalaSQL on *hdpserver*** dialog appears, ensure that your username and password are correct.
- ☐ Click the Connect button.
- ☐ If all goes well, SQuirreL should connect to the Hadoop database via Impala and a tab should appear containing the tabs Objects, SQL and Hibernate.
- ☐ At this point, your schema will probably be devoid of tables, so there's nothing much to do.
- ☐ Exit out of SQuirreL.
- ☐ As an additional test, try to connect to the Hadoop database using the **HiveQL on *hdpserver*** alias.

Oracle SQL Developer

Oracle SQL Developer is a free integrated development environment (IDE) for querying Oracle and MySQL. Despite its name, Oracle SQL Developer can also connect to a Hadoop database, but at this point can only connect via Hive, not Impala.

Please create a folder labeled `OracleSQLDeveloper` under `\\corp\dept\software`.

Let's download and install Oracle SQL Developer.

- ☐ Navigate to `https://www.oracle.com/database/technologies/appdev/sqldeveloper-landing.html`.
- ☐ Under the Oracle SQL Developer section, click on the SQL Developer button.
- ☐ Click the Download link to the right of the text **Windows 64-bit with JDK 8 included**.
- ☐ Accept the license agreement when the popup dialog box…er…pops up and then click **Download** `sqldeveloper-#.#.#.#.#-x64.zip`.
- ☐ You will need to sign in to your Oracle account or create a new one. Follow the instructions to the letter or someone from Oracle will come to your house and stare at you angrily.
- ☐ Once the download completes, move it to the `OracleSQLDeveloper` folder.
- ☐ To install the software is only a matter of unzipping using WinZip (or other utility). Navigate to the file `sqldeveloper-#.#.#.#.#-x64.zip` and double-click on it to start WinZip.
- ☐ Once WinZip starts, you should see a folder named `sqldeveloper`. (Good thing because if the folder were named `CIA Secret Documents` I'd be real worried like!) Drag this folder to a location on your laptop. I usually just put it under `C:\TEMP`.

Next, let's start Oracle SQL Developer and set up a connection to Hive.

- ☐ Navigate into the `sqldeveloper` folder and you'll see an executable named `sqldeveloper.exe`. (Good thing because…oh, nevermind!) Double-click the executable to start Oracle SQL Developer.
- ☐ When (or If) the **Oracle Usage Tracking** dialog box appears, ensure that the checkbox to the left of the text **Allow automated usage reporting to Oracle** is unchecked.
- ☐ Click OK.
- ☐ Before setting up Hive, we need to indicate to Oracle SQL Developer where the Java `.jar` files are located. Click on Tools…Preferences.
- ☐ When the **Preferences** dialog box appears, expand the **Database** branch and click on the **Third Party JDBC Drivers** leaf.
- ☐ Click on the Add Entry… button.
- ☐ Navigate to where you placed the `HiveJDBC41.jar` file, highlight it and click Select.
- ☐ Click on the Add Entry… button (once again).
- ☐ Navigate to where you placed all of the Java `.jar` files associated with Impala. Expand that folder and highlight all of the JAR files and click Select.
- ☐ Click OK.
- ☐ To create a new connection, click the green plus-sign in the Connections pane on the left.
- ☐ When the **New/Select Database Connection** dialog box appears, select the Hive entry from the drop-down box labeled **Database Type**.
- ☐ Fill in the entries like this:
    - Name: Hive on *hdpserver*
    - Username: <*enter your username*>
    - Password: <*enter your password*>
    - Ensure the checkbox to the left of the text **Save Password** is checked.
    - Host name: <*hdpserver*>
    - Port: 10000
    - Database: *prod_schema*
    - Driver: CLOUDERA HIVE JDBC 4.1
    - Click the Add button, highlight `AuthMech`, then click OK.
    - Select `3` from the AuthMech drop-down box.
- ☐ Click Save to save the connection.
- ☐ Click Test to test the connection to the database. If all goes well, you will see the nearly invisible text `Success` appear on the lower-left hand corner of the dialog box. If not, update your connection information and try again.

FileZilla

FileZilla is a free, open source FTP/SFTP client you can use to transfer files between your laptop and the Linux edge node server – as we indicated in *Chapter 1 – Quick Start Guide*. Let's download and install FileZilla.

Please create a folder labeled `FileZilla` under `\\corp\dept\software`.

- ☐ Navigate to `https://filezilla-project.org/`.
- ☐ Click on the button labeled Download FileZilla Client All Platforms.
- ☐ On the **Download FileZilla Client** webpage, click the **Download FileZilla Client** button under Windows (64bit x86).
- ☐ Once the download completes, move the file `FileZilla_#.#.#_win64_sponsored-setup.exe` to the `FileZilla` folder.
- ☐ To install the software, double-click `FileZilla_#.#.#_win64_sponsored-setup.exe`.
- ☐ When the **License Agreement** dialog box appears, click on the I Agree button.
- ☐ On the **Optional Offer** dialog box, ensure the radio button to the left of the text **Decline** is selected and click the Next button.
- ☐ On the **Choose Installation Options** dialog box, ensure the radio button to the left of the text **Only for me (BobSmith)** is selected and click the Next button.
- ☐ On the **Choose Components** dialog box, ensure all checkboxes are checked and click the Next button.
- ☐ On the **Choose Install Location** dialog box, click the Next button.
- ☐ On the **Choose Start Menu Folder** dialog box, click the Install button.

- ☐ The software may take some time to install, so it's a good time to learn how to make your own shoes in preparation for the ensuing apocalypse.
- ☐ On the **Completing FileZilla Client #.#.# Setup**, ensure the checkbox to the left of the text **Start FileZilla now** is checked and click the Finish button.
- ☐ Once FileZilla starts, we can set up a connection to the Linux edge node server (`lnxserver`).
- ☐ Open the FileZilla Site Manager by clicking on the Open the Site Manager icon on the far left just above the text Host.
- ☐ Click the New Site button.
- ☐ On the left side, enter a name for the site such as `lnxserver as bobsmith`.
- ☐ On the right side, fill in the dialog like this:
    - Protocol: SFTP
    - Host: *lnxserver*
    - Port: 22
    - Logon Type: Normal
    - User: *<enter your username>*
    - Password: *<enter your password>*
- ☐ Click the OK button to save your entries.

Next, let's log into the Linux edge node server.

- ☐ Click the Site Manager button to bring up the Site Manager.
- ☐ Highlight the entry labeled `lnxserver as bobsmith`.
- ☐ Click the Connect button.
- ☐ After the briefest of pauses, FileZilla will connect to the Linux server. At this point you can transfer files between your laptop and the Linux server by dragging the files between the left and right panes. The left pane represents your laptop hard drive while the right pane represents the Linux server located initially in the `/home/smithbob` folder.

WinSCP

WinSCP, like FileZilla, is a free FTP/SFTP client. You use WinSCP to transfer files between your laptop and the Linux edge node server, as we saw in *Chapter 1 – Quick Start Guide*. Let's download and install WinSCP.

Please create a folder labeled `WinSCP` under `\\corp\dept\software`.

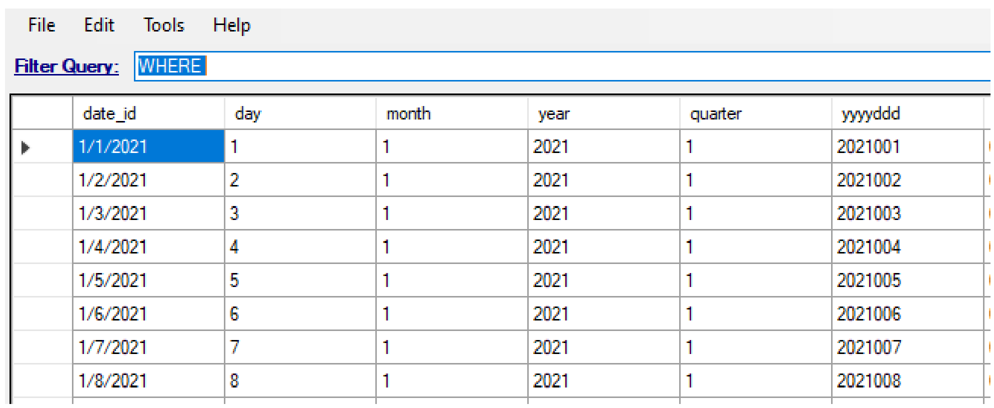
- ☐ Navigate to `https://winscp.net/`.
- ☐ Click on the DOWNLOAD NOW button.
- ☐ Click on the DOWNLOAD WINSCP #.#.# (# MB) button.
- ☐ When the download finishes, move the file `WinSCP-#.#.#-Setup.exe` to the `WinSCP` folder.
- ☐ Double-click on the file `WinSCP-#.#.#-Setup.exe` to the `WinSCP` folder.
- ☐ On the **Select Setup Install Mode** dialog box, click on **Install for me only**.
- ☐ On the **License Agreement** dialog box, click on the Accept button.
- ☐ On the **Setup Type** dialog box, ensure the radio button to the left of the text Typical installation (recommended) is selected and then click Next.
- ☐ On the **Initial User Setting** dialog box, ensure the radio button to the left of the text Commander is selected and then click Next.
- ☐ On the **Ready to Install** dialog box, click the Install button.
- ☐ As part of the WinSCP installation, it nosily looks for sites in PuTTY as well as FileZilla. *Bad WinSCP! Bad!* When the **Confirm – WinSCP** dialog box appears, click on Yes to import these sites.
- ☐ On the **Import sites – WinSCP** dialog box, click OK.
- ☐ On the Completing the WinSCP Setup Wizard dialog box, leave the two checkboxes checked and click the Finish button.
- ☐ When WinSCP appears, you should see an entry for `lnxserver as bobsmith`. Double-click on that entry to test the login.
- ☐ Once logged in, similar to FileZilla, you'll see your laptop hard drive on the left and the remote Linux server on the right.

Parquet Viewer

You may occasionally want to view the data held in a Parquet-formatted file. We discuss the Parquet format later in the book. A free Windows option is ParquetViewer, located at `https://github.com/mukunku/ParquetViewer` and we discuss this further in *Chapter 38 – The parquet-tools and parquet-cli Utilities*.

Please create a folder labeled `ParquetViewer` under `\\corp\dept\software`.

- ☐ Navigate to `https://github.com/mukunku/ParquetViewer`.
- ☐ Scroll down to the **Download** section.
- ☐ Click on the **Pre-compiled releases** link.
- ☐ Under the **Assets** section for the latest release (v2.3.6 as of this writing), right-click over `ParquetViewer.exe` and save it to the folder.
- ☐ Note that this executable is not an installer, but the application itself. Run this executable through your anti-virus software.
- ☐ Right-click over the executable and click Properties. Ensure the checkbox to the left of the text **Unblock** is checked and click OK to dismiss the dialog box.
- ☐ Copy your Parquet file from HDFS to the Linux filesystem in the directory, say, `/home/smithbob`. ParquetViewer expects the extension to be `.parquet`, so rename your file if necessary. FTP this file to your Windows laptop.
- ☐ For example, here's the `DIM_CALENDAR` table we create later on in the book displayed in ParquetViewer:



File Edit Tools Help

Filter Query: WHERE

date_id	day	month	year	quarter	yyyyddd
1/1/2021	1	1	2021	1	2021001
1/2/2021	2	1	2021	1	2021002
1/3/2021	3	1	2021	1	2021003
1/4/2021	4	1	2021	1	2021004
1/5/2021	5	1	2021	1	2021005
1/6/2021	6	1	2021	1	2021006
1/7/2021	7	1	2021	1	2021007
1/8/2021	8	1	2021	1	2021008

- ☐ A nice feature is the `Filter Query` inputbox which allows you to place a `WHERE` Clause used to subset the data. Nice!!

Note that there's the Linux command line utility `parquet-tools` which allows you to view data (as well as a variety of other information) stored in a Parquet file. We talk more about the Parquet format as well as `parquet-tools` later in the book.

Chapter 4 – A Teensy-Weensy Chat about Hadoop

In this chapter, we briefly discuss Hadoop in very simplified terms. As I mentioned earlier, this is not a book about the intricacies of Hadoop. But, still, you need to know something to impress people at cocktail parties, right?

In the Beginning…

Traditional databases, such as Oracle, SQL Server, Teradata, etc., have a *my-way-or-the-highway* attitude; that is, how their databases operate under the hood is proprietary: Oracle stores its data in its own format; SQL Server, its own format; Teradata, its own format. And knowing what these formats are is not your business, puny human!

Even accessing these databases is usually via their own proprietary SQL query tools – SQL*Plus for Oracle, SQL Server Management Studio for SQL Server, SQL Assistant for Teradata, etc. Naturally, you can set up ODBC connections to these databases as well as use other SQL clients, but in the end, you're connecting to a database-proprietary process which executes your SQL query. Just imagine if Jim-Jane-Jake Corporation invented a faster way to process SQL queries by accessing Oracle's own underlying proprietary data format…Jim and Jane would be laughed outta Oracle's headquarters, while Jake would be hurled off the roof. Splat!

But, with Hadoop, it's a different story. Now, despite the zillions of lines of code used to execute queries, manage database tables, track thousands of files across countless hard drives, and generally keep the entire system from collapsing like a badly timed soufflé, Hadoop comes down to one extremely important feature: the Hadoop Distributed File System, or HDFS. (In the previous chapters, I used the word *Hadoop* as a temporary stand-in for *HDFS*.) A *file system* is used to keep track of files, similar to the files on your laptop's hard drive. On Windows, the file system is called the New Technology File System, or NTFS; on Linux, it's called the ext4 journaling file system; on floppy disks, it's called the File Allocation Table, or FAT; and so on. It's the job of the file system to respond "*Yo! Over here, dude!*" when asked by the operating system where a file - or portion of a file (*insert ominous music here!*) - is located.

This is true of HDFS as well. As you saw in *Chapter 1 – Quick Start Guide*, we copied a file from the Linux file system under the directory `/home/smithbob` to a directory in HDFS where Hadoop was able to access the file. If we didn't do this, Hadoop wouldn't see the file…game over! This means that the Linux file system and the Hadoop Distributed File System (HDFS) are separate entities. In other words, Hadoop has reign over the Hadoop Distributed File System whereas Linux has reign over the Linux file system…and never the twain shall meet. (This is a slight lie, but go with it, it's brilliant!)

Now, above I used the ominous phrase "…or a portion of a file…". So, what does that mean? On your laptop hard drive, let's say you download an alien space battle game called `alienspacebattlegame.exe`. The file system will store this file on the hard drive either as one contiguous piece, or as several smaller pieces (or fragments) with pointers between the pieces. In either case, the operating system can execute the game successfully (though your success at actually playing the game is a different story).

In HDFS, when you create a table, depending on the size, Hadoop may break apart the table into smaller, more manageable file pieces. These smaller pieces are spread across the many hard drives used by HDFS. This then allows any SQL query on that table to run faster because several processes can run simultaneously on different portions of the file and across several servers as well. At the end, the results are combined and you're presented with a complete query result, brand new table loaded with your fab data, etc. etc.

Recall from *Chapter 1 – Quick Start Guide* that we used the Hadoop command `getmerge` to create a file for Big Mike the Sales Troll. It's possible that the underlying table, `prod_schema.bigmike_output`, was broken up into pieces and spread across HDFS, like my body parts on a *Call of Duty* map ☹. By using `getmerge`, all of these pieces are combined into one large file which we named `bigmike_output.tsv`.

For example, from the ImpalaSQL command line (accessible from the `impala-shell` utility), we can list all of the files that comprise the table `prod_schema.dim_postal_code`:

```
[hdpserver:21000] prod_schema> show files in prod_schema.dim_postal_code;
+----------------------------------------------------------------------------------------------------------------------+----------+
| Path                                                                                                                 | Size     |
+----------------------------------------------------------------------------------------------------------------------+----------+
| hdfs://hdpserver/data/prod/teams/prod_schema/dim_postal_code/a04dfbd11c2688bf-f6b48dc300000000_1117506544_data.0.parq | 132.33MB |
| hdfs://hdpserver/data/prod/teams/prod_schema/dim_postal_code/a04dfbd11c2688bf-f6b48dc300000001_1011983093_data.0.parq | 24.68MB  |
+----------------------------------------------------------------------------------------------------------------------+----------+
```

As you see, the table is comprised of two individual files on disk which make up the entire table. And because that table is a managed table, the file names have been assigned by Hadoop automatically (which is why they look like a Martian high schooler's locker combination). We talk more about managed and external tables in the next section and in *Chapter 23 – Working with Managed and External Tables*.

Besides spreading file pieces across HDFS, depending how your administrator set up Hadoop, each piece may be *replicated* many times so that if a hard drive failure occurs – which only happens to other people ☺ – your SQL queries will run uninterrupted. Sweet!!

Who's Managing (or Not Managing) My Tables?

When you create a table – or do any other operation on that table – in a traditional database such as Oracle, SQL Server, Teradata, etc., there's no doubt in your mind that the database lovingly manages everything for you behind the scenes. Never do you have to go into ***HULK SMASH!*** mode when you create a table…it just works. *Bingo!*

This type of table is called a *managed table* because the database…uh…manages the table for you.

Now, some traditional databases allow access to data files stored on disk outside the control of the database itself. For Oracle and SQL Server, these data files can be accessed using the `ORGANIZATION EXTERNAL` Clause on the `CREATE TABLE` Statement. For Teradata, it's known as a foreign table and can be accessed using the `CREATE FOREIGN TABLE` Statement. In these cases, you have more control because you're telling the database exactly where the data files are located on disk, the field delimiter, the row delimiter, among other things.

This type of table is called an *external table* and you have more control over it as compared to a managed table.

And, as you've probably guessed already, Hadoop allows for both *managed tables* and *external tables*. When you use the `CREATE TABLE` Statement, you're creating a *managed table* that the Hadoop database will handle for you. When you use the `CREATE` **`EXTERNAL`** `TABLE` Statement, you're pointing to a **location** in HDFS where your file or files are stored. (This is a slight lie, but go with it, it's brilliant!)

Recall that in *Chapter 1 – Quick Start Guide*, we created an external table to read in the US state codes and names. We also did the reverse tactic to create that file for Big Mike the Sales Shmuck. In both cases, you have more control over the file location, field separator (e.g., comma, tab, etc.), additional options such as *ignore number of header lines* or *replace NULL with a blank*, etc. But, recall that we created the final table `prod_schema.dim_postal_code` using a `CREATE TABLE` Statement. This table is a managed table, not an external table, because the keyword `EXTERNAL` has been left off the `CREATE TABLE` Statement.

Now, I can hear y'all say, "*Hang on! Surely, for traditional databases as well as Hadoop, don't all tables eventually reside somewhere on disk?*" Yes, this is true! And don't call me Shirley. For both traditional databases as well as Hadoop, table data resides on disk somewhere. Someone once equated a database to a grown-up file server, but don't say that to a DBA because they become irate and age quickly (it's not the tipple, as some suppose). Similar to external tables, managed tables reside on disk except Hadoop handles the file location, creates the names of the underlying files, manages the format of the files (*insert really ominous music here!*), etc.

To determine if a table is *managed* or *external*, you can use the `DESCRIBE FORMATTED` Statement and observe the row labeled **Table Type**. For example, `desc formatted prod_schema.dim_postal_code;` produces the output below. Take note that the Table Type is `MANAGED_TABLE`.

```
[hdpserver:21000] prod_schema> desc formatted dim_postal_code;
+------------------------------+------------------------------------------------------------+----------------------+
| name                         | type                                                       | comment              |
+------------------------------+------------------------------------------------------------+----------------------+
| # col_name                   | data_type                                                  | comment              |
|                              | NULL                                                       | NULL                 |
| postal_code                  | string                                                     | NULL                 |
| city                         | string                                                     | NULL                 |
| state_code                   | string                                                     | NULL                 |
| latitude                     | double                                                     | NULL                 |
| longitude                    | double                                                     | NULL                 |
|                              | NULL                                                       | NULL                 |
| # Detailed Table Information | NULL                                                       | NULL                 |
| Database:                    | prod_schema                                                | NULL                 |
| OwnerType:                   | USER                                                       | NULL                 |
| Owner:                       | smithbob                                                   | NULL                 |
| CreateTime:                  | Thu Sep 09 09:35:23 CDT 2021                               | NULL                 |
| LastAccessTime:              | UNKNOWN                                                    | NULL                 |
| Retention:                   | 0                                                          | NULL                 |
| Location:                    | hdfs://hdpserver/data/prod/teams/prod_schema/dim_postal_code| NULL                |
| Table Type:                  | MANAGED_TABLE                                              | NULL                 |
| Table Parameters:            | NULL                                                       | NULL                 |
|                              | transient_lastDdlTime                                      | 1631198123           |
|                              | NULL                                                       | NULL                 |
| # Storage Information        | NULL                                                       | NULL                 |
| SerDe Library:               | org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe         | NULL                 |
| InputFormat:                 | org.apache.hadoop.mapred.TextInputFormat                   | NULL                 |
| OutputFormat:                | org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat | NULL                 |
| Compressed:                  | No                                                         | NULL                 |
| Num Buckets:                 | 0                                                          | NULL                 |
| Bucket Columns:              | []                                                         | NULL                 |
| Sort Columns:                | []                                                         | NULL                 |
+------------------------------+------------------------------------------------------------+----------------------+
```

Let's do that again, but this time for the external file we produced for Big Mike the Sales Trollop:

```
[hdpserver:21000] prod_schema> desc formatted bigmike_output;
+------------------------------+------------------------------------------------------------+----------------------+
| name                         | type                                                       | comment              |
+------------------------------+------------------------------------------------------------+----------------------+
| # col_name                   | data_type                                                  | comment              |
|                              | NULL                                                       | NULL                 |
| postal_code                  | string                                                     | NULL                 |
| city                         | string                                                     | NULL                 |
| state_code                   | string                                                     | NULL                 |
| latitude                     | double                                                     | NULL                 |
| longitude                    | double                                                     | NULL                 |
| state_name                   | string                                                     | NULL                 |
|                              | NULL                                                       | NULL                 |
| # Detailed Table Information | NULL                                                       | NULL                 |
| Database:                    | prod_schema                                                | NULL                 |
| OwnerType:                   | USER                                                       | NULL                 |
| Owner:                       | smithbob                                                   | NULL                 |
| CreateTime:                  | Thu Sep 09 13:38:59 CDT 2021                               | NULL                 |
| LastAccessTime:              | UNKNOWN                                                    | NULL                 |
| Retention:                   | 0                                                          | NULL                 |
| Location:                    | hdfs://hdpserver/data/prod/teams/prod_schema/bigmike_output| NULL                 |
| Table Type:                  | EXTERNAL_TABLE                                             | NULL                 |
| Table Parameters:            | NULL                                                       | NULL                 |
|                              | EXTERNAL                                                   | TRUE                 |
|                              | OBJCAPABILITIES                                            | EXTREAD,EXTWRITE     |
|                              | serialization.null.format                                  |                      |
|                              | transient_lastDdlTime                                      | 1631212739           |
|                              | NULL                                                       | NULL                 |
| # Storage Information        | NULL                                                       | NULL                 |
| SerDe Library:               | org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe         | NULL                 |
| InputFormat:                 | org.apache.hadoop.mapred.TextInputFormat                   | NULL                 |
| OutputFormat:                | org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat | NULL                 |
| Compressed:                  | No                                                         | NULL                 |
| Num Buckets:                 | 0                                                          | NULL                 |
| Bucket Columns:              | []                                                         | NULL                 |
| Sort Columns:                | []                                                         | NULL                 |
| Storage Desc Params:         | NULL                                                       | NULL                 |
|                              | field.delim                                                | \t                   |
|                              | serialization.format                                       | \t                   |
+------------------------------+------------------------------------------------------------+----------------------+
```

Take note that the Table Type is `EXTERNAL_TABLE` this time.

What's the (For)matter, Buddy?

In the last section, I really ominously indicated that Hadoop manages the *format of the files* in HDFS. So, what does that mean? Well, unlike traditional databases, whose underlying file format is proprietary, Hadoop allows you to choose in which format you'd like your table data to be stored. This is called the table's *storage format*. This is a really weird concept coming from a traditional database background in which everything is handled for you behind the scenes.

As indicated earlier, whenever you create a table, you're actually creating a directory in HDFS which is used to store your table's data. This data can be *read from* using SQL, and can be *written to* when either `CREATE TABLE AS` or `INSERT` is executed. In order to carry out both of these file-related operations – *reading from* and *writing to* – Hadoop associates one Java class responsible for *reading from* and another Java class responsible for *writing to*. The Java class used to *read from* disk is referred to as its *input format* whereas the Java class used to *write to* disk is referred to as its *output format*.

There are several storage formats available for you to use and we've already seen one of them in *Chapter 1 – Quick Start Guide*. Recall that we used the clause `STORED AS TEXTFILE` when creating the comma-delimited file for Big Mike the Sales Ho'. This indicates that we want the data to be stored just like a regular ol' text file. We used the keyword `TEXTFILE` to indicate this on the `STORED AS` Clause. There are several other storage formats and some of the more star-studded biggies are `PARQUET` and `KUDU`.

But, the `STORED AS` *`storage-format`* Clause is just short-hand notation saving you the trouble of specifying the Java classes that handle input and output activities, as well as a third Java class which handles *working with* the data. The keywords for these three activities are, respectively, *input format*, *output format* and *serde*. The *input format* and *output format*, as described above, are responsible for file-related activities such as *reading from* and *writing to*, while *serde* is responsible *working with* the rows of data in a table. This is an over-simplification and we explain it in more detail in *Chapter 23 – Working with Managed and External Tables*.

If this is confusing, you can think of these three activities like this: When you tell Microsoft Excel to open up a delimited text file, Excel displays a dialog box asking you to indicate how the file is delimited. Once Excel knows how the file is delimited, it can proceed with its *input format* activity; that is, parsing the file based on the specified delimiter. Next, Excel then has to analyze each field to determine if it's a text string, a number, or a date. This is Excel's *serde* activity which allows Excel to populate the worksheet with the data from the delimited file. Finally, when you save the Excel workbook to disk in XLSX or other format, Excel is performing its *output format* activity.

For example, above we described the table `prod_schema.bigmike_output` to see the Table Type, but near the end of the output you'll notice three lines indicating how the table data is actually stored:

```
| SerDe Library:            | org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe         | NULL                 |
| InputFormat:              | org.apache.hadoop.mapred.TextInputFormat                   | NULL                 |
| OutputFormat:             | org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat | NULL                 |
```

The Java class `LazySimpleSerDe` is the *serde*, `TextInputFormat` is the *input format* and `HiveIgnoreKeyTextOutputFormat` is the *output format*. All three taken together indicate that the table is `STORED AS TEXTFILE`. Note that not all available storage formats have a corresponding simplified option using the `STORED AS` Clause. This makes us sad! In those cases, the use of the *input format*, *output format* and *serde* clauses is necessary. We discuss this more in *Chapter 23 – Working with Managed and External Tables*.

As mentioned above, each of the three storage types `TEXTFILE`, `PARQUET`, and `KUDU` are just short-hand notations saving you from the drudgery of specifying the Java classes for *input format*, *output format* and *serde*. So, why are these three storage formats special? Well, they aren't. There are other storage formats that can be used with the `STORED AS` Clause: `AVRO`, `ORC`, `SEQUENCEFILE`, `RCFILE` and `JSONFILE`. And, in the future, there will be more added to the list. (`ORC`…that's a funny word!)

At this point, you're probably trolling the Internet for new job opportunities…*RUN AWAY!*…*RUN AWAY!* But, wait! You can limit yourself to just three of the storage formats: `TEXTFILE`, `PARQUET` and `KUDU`. My team and I only use these three formats, not the others, and things are working great. Here's a more detailed description of them:

1. `TEXTFILE` – While not the most efficient in terms of disk space and not the fastest storage format on Broadway, it allows you to *read from* text files as well as *write to* text files. If you don't specify the `STORED AS` Clause on the `CREATE [EXTERNAL] TABLE` Statement, `TEXTFILE` is usually the default.
2. `PARQUET` – This storage format makes better use of space and is faster due to its compression and internal data organization. This is the storage format you'll be using most often.
3. `KUDU` – This storage format allows you to perform `UPDATE`s and `DELETE`s on your table (*insert really, really ominous music here!!*). This is the storage format you'll probably be using second most often.

Give It To Me Straight, Doc!

In the last section, I really, really ominously indicated that the `KUDU` storage format allows you to perform `UPDATE`s and `DELETE`s on tables created with the `KUDU` storage type. This begs the question: Do the storage formats `TEXTFILE` and `PARQUET` allow for `UPDATE`s and `DELETE`s? I'm gonna tell ya', **BUT YOU HAVE TO PROMISE NOT TO PUNCH ME!**

Both `TEXTFILE` and `PARQUET` do not allow for `UPDATE`s and `DELETE`s.

NO!! NO!! NOT IN THE FACE!!

(Again, this is a slight lie, but go with it, it's brilliant!)

Although your legacy database allows for `UPDATE`s and `DELETE`s on its managed tables, the loss of these statements is not as drastic as you think, especially for a data warehouse. Those tables that are considered read-only tables should be given a storage type of `PARQUET`, whereas those tables that need `UPDATE`/`DELETE` capabilities during the regular course of business should be given the storage format of `KUDU`. If a `PARQUET` table ever needs `UPDATE`s or `DELETE`s performed on it, you either can perform these statements in the legacy database and `sqoop` the altered table down to the Hadoop database; or, you can modify the table in the Hadoop database by first creating a `KUDU` table from the `PARQUET` table, perform the `UPDATE`s and `DELETE`s and then re-create the table as `PARQUET`; or, just bite the bullet and create the table as `KUDU` and perform `UPDATE`s and `DELETE`s at will. We've never seen a drastic performance hit between `PARQUET` and `KUDU`, but it's best to use `KUDU` only when you really need `UPDATE` and `DELETE` capabilities; otherwise, it's a waste to use that storage format.

The Two Elephants in the Room: HiveQL and ImpalaSQL

I mentioned both the Impala and Hive query engines earlier in the book and indicated that people may think there's a battle raging between them. It's probably a good idea to explain why this might be.

In order to make the conversion from your legacy database to the Hadoop database as seamless as possible, it's best to use a query language that closely matches the SQL in your legacy database. Both HiveQL (the SQL available in the Hive query engine) and ImpalaSQL (the SQL available in the Impala query engine) match very closely with legacy database SQL with minor exceptions. For example, the `INTERSECT` and `MINUS`/`EXCEPT` operators may not be available, but these can be easily replaced by standard SQL code. And the newer extensions to the `GROUP BY` Statement, such as `GROUP BY CUBE`, `GROUP BY ROLLUP` and `GROUP BY GROUPING SETS` may not be available, but can be created either with `UNION`s or by creating a procedure using HPL/SQL.

Now, there are several other query languages available for you to use against the Hadoop database, such as Pig, Apache Spark, etc. But, these don't necessarily use the standard SQL syntax we've all become accustomed to and trying to train your team members on a new query language during the conversion is fraught with problems. Also, trying to maintain your system's integrity while queries are written in multiple languages, some of which aren't known by the majority of your team members, is just a problem waiting to happen. I don't mean that your team members should be prevented from learning these other tools, quite the contrary, but you need to think about the entire system before adding a new query language into the mix: Is it truly beneficial or is it just something fun to learn?

I can hear you thinking: *So, if HiveQL is the SQL query language for Hive, and ImpalaSQL is the SQL query language for Impala…what the hell are Hive and Impala then?*

I like to think of Hive and Impala as Lego blocks that click into Apache Hadoop's foundation making use of its many pre-existing software components to interact with HDFS, query vast amounts of data, etc. In the olden days, querying Apache Hadoop was via gut-busting Java APIs. Apache Hive (Hive, to the cognoscenti) sits on top of Apache Hadoop and makes the SQL-like language HiveQL available to easily query a Hadoop database…and SQL programmers around the world rejoiced! Apache Impala (Impala, to some other cognoscenti) also sits on top of Apache Hadoop, but whereas Apache Hive provides HiveQL, Apache Impala goes even further by providing a zippy query engine with the ability to process queries in parallel via its own SQL-like language ImpalaSQL. Yes, it's Jim-Jane-Jake Corporation to the rescue (all except Jake…he's dead).

With that explanation out of the way, we've found that Hive is a real porker…it's very slow! On the other hand, we've found that Impala is extremely fast. ImpalaSQL is the query language my entire team uses almost exclusively. With that said, I use HiveQL occasionally because there are additional Java classes available for use with *input format*, *output format* and *serde* that aren't available in ImpalaSQL. These additional Java classes allow you to read in more complex files. Since most of your team members, I assume, will be querying the database and not loading in data, HiveQL should probably take a back seat to ImpalaSQL for those team members, at least until your conversion is complete and things are running smoothly. Be aware, though, that for very large SQL queries, Impala can run out of server resources whereas Hive would, mostly likely, be able to complete the query (although much slower).

So…Hive and Impala Play Nice?

Usually, Hive and Impala play nice together, but there are a few things you should know. Since many of your database users will be executing ImpalaSQL exclusively, they won't need to worry (too much) about these issues.

When using PuTTY to access the command line on the Linux edge node server, you run HiveQL queries using the utility `beeline` (formerly known as `hive`, which has been defecated…er…deprecated) and run ImpalaSQL queries using the utility `impala-shell`. Alternatively, you can use a SQL client to connect either to Hive or Impala to run SQL queries. In either case, if you create a table in Hive, Impala doesn't immediately acknowledge that the table exists. In order for Impala to see the table, you run the following command in ImpalaSQL right after you create the table in HiveQL:

```
INVALIDATE METADATA PROD_SCHEMA.MY_NEW_TABLE;
```

At this point, you'll be able to access the table as if it were created natively in Impala using ImpalaSQL. But…

So…Hive and Impala Play Nice (REDUX)?

Note that Impala doesn't recognize every storage format (*input format*, *output format* and *serde*) available in Hive. Recall I mentioned I use Hive occasionally to read certain file formats which Impala cannot read. If you create a table in Hive using one of these storage formats, `INVALIDATE METADATA` and then try to query the table in Impala, you'll be greeted with big honkin' error message indicating that Impala cannot access the table data. The workaround is to create, say, a `PARQUET` table in Hive after you've read in the data using the Hive-specific *input format* format, run `INVALIDATE METADATA` on the table in Impala and you'll be good to go: Impala recognizes the `PARQUET` format. Get it? With that said, the `KUDU` storage format is still currently not recognized by Hive.

The Author's a Big Fat Liar!

At several points in my droning explanations above, I mentioned I was lying. Let's try to make amends here:

- ☐ I mentioned above that Hadoop reigns supreme over HDFS and Linux reigns supreme over the Linux file system…and never the twain shall meet. This is not quite true. You can, in fact, use the ImpalaSQL `LOAD DATA` command to push data files located in a Linux subdirectory directly into HDFS. I don't often use this

command since it deletes the files in the Linux subdirectory…and that gives me violent stomach cramps. We discuss `LOAD DATA` in *Chapter 30 – Loading Data using LOAD DATA to Load Data*. Also, there are methods to mount HDFS from the Linux filesystem, but that's not necessary for what we're trying to accomplish in this book.

- ☐ I mentioned above that when you use the `CREATE TABLE` Statement, you're creating a managed table that the Hadoop database will handle for you; and, when you use the `CREATE EXTERNAL TABLE` Statement, you're pointing to a location in HDFS where your file(s) are stored. This is not exactly true all of the time. Your administrator may set up Hadoop so that managed tables exhibit external table behavior. This may be due to several factors and I'll just leave it at that. From a SQL programming perspective, you won't notice the difference.
- ☐ I mentioned above that both the `TEXTFILE` and `PARQUET` storage formats do not allow for `UPDATE`s and `DELETE`s. In later versions of Apache Hadoop, you can perform `UPDATE`s and `DELETE`s on tables created using these storage formats, but you're required to set an option on the `CREATE TABLE` Statement. Note that caution is advised since this is a fairly new option, whereas `KUDU` has been around for donkey's years. Honestly, we've never had any serious problems with `KUDU`, so give that storage format a whirl, buddy!
- ☐ Beginning with Hive version 3, managed tables can be used for ACID (atomicity, consistency, isolation, durability) transactions which ensures data validity in the pimply face of data errors, server crashes, etc.
- ☐ Although not mentioned – so I guess it's a lie by omission – the HDFS directory for *managed* tables is different from the HDFS directory for *external* tables. Recall I mentioned that the directory in HDFS for your team may be named `/data/prod/teams/prod_schema`. When creating managed tables, the directory in HDFS is under Hadoop's control, so you don't have to worry about it, but it won't be `/data/prod/teams/prod_schema`. When creating external tables, you'll most likely use some other directory and recall that one of the requests in the Hadoop Administrator E-Mail is to create a directory in HDFS for your team to use. This directory is, most likely, specifically for your external tables, not your managed tables.

Hang On! Did You Say Non-Edge Node Servers?

Earlier I mentioned that *An edge node server is a Linux server that is* ***not*** *one of the Hadoop servers used to process SQL queries*. Sure, it's a crap sentence, but it's all mine! Now, Hadoop uses a variety of servers (aka, *nodes*) to execute SQL queries as well as manage itself:

1. NameNode – You can think of this server as *air traffic control* for the entire Hadoop *airspace*. If this server is down, there's gonna be a lotta trouble, Lucy! Similar to the file system on your laptop's operating system, this server tracks the locations of the files stored across the entire cluster. The NameNode is the master in the *NameNode-DataNodes* relationship and, as such, assigns work to the DataNodes. The NameNode is in constant communication with the DataNodes as queries progress, as additional files are added, as data is being flung around the cluster, and so on. Note that the your multidimensional Hadoop Administrator keeps multiple NameNodes in a high availability state to avoid a complete database meltdown when one of the NameNodes crashes.
2. DataNode – These servers are responsible for storing the actual files in HDFS. Recall I mentioned earlier that Hadoop replicates the data. If a DataNode is down, the NameNode can usually workaround this issue because replicas of the data are stored on other DataNodes. In this case, Lucy won't be in trouble. You can think of DataNodes as slaves or workers to the NameNode master.

As you can well imagine, there's a lot more going on with Hadoop than this chapter's mediocre and run-on-sentence-infused explanations have exposed. Please see *Appendage #5 – Where Do I Go from Here?* for recommended books to peruse and websites to visit.

Chapter 5 – Creating Your Very Own Hadoop Playground

After reading the last chapter, you're probably all a-tingle and a-sweaty with excitement about this Hadoop shizz. In this chapter, I'll show you how to create your own Hadoop Playground on your Windows laptop by downloading and installing a variety of software, pushing a few buttons, decanting some incantations, etc.

Note that there are additional ways to go here, namely to sign up for Cloudera's Public Cloud, Amazon Web Services, Microsoft Azure, etc., but some of these services require giving away your creditcard information as well as your first born child…thpppttt! Just like marriage, you're limited to a certain "free period" before you must shell out your hard-earned simoleons.

There are several Hadoop suites available on the InterWebs for free, some require virtual machine software (such as VMWare or VirtualBox), and others require Docker to be installed. We discuss this throughout the rest of this chapter.

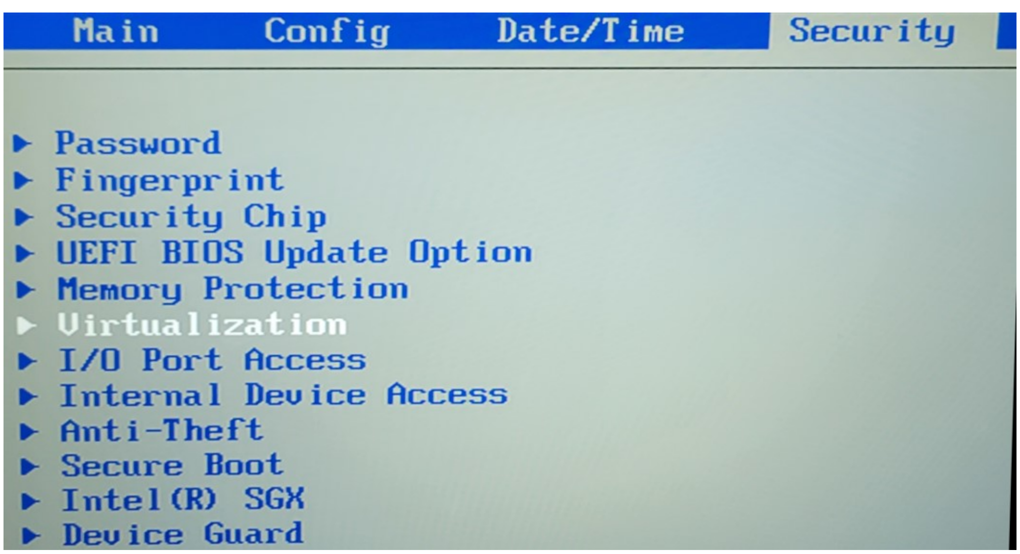
Note that much of the freely available software out there is a version or two behind the current versions. That should be fine for learning/testing purposes, but your job will most likely have the latest versions available to play with anyway. In that case, you should probably *eschew* the content below and *chew* on the company servers instead.

Virtualization and the BIOS

In order for virtual machine software, such as VMWare and VirtualBox, to function properly, your laptop's BIOS virtualization setting needs to be switched on. **Before performing this step, ensure that you've backed up all important files, or you've created a complete image of your laptop using CloneZilla, Macrium Reflect, or other software.**

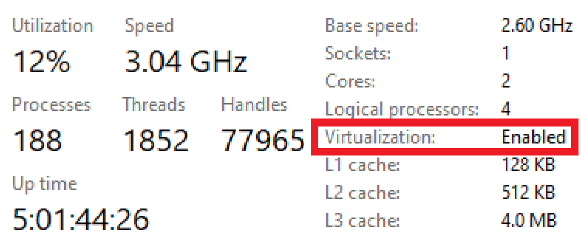
To turn on virtualization in your laptop's BIOS, perform the following steps:

- ☐ Restart your laptop.
- ☐ Based on the make/model of your laptop, go into the BIOS by pressing the indicated button(s). On some laptops, you press the Delete key several times. On other laptops, you press the F12 button. Again, it depends on the make/model of your laptop, so google away, pal!
- ☐ Once you're in the BIOS's main screen, you'll have to navigate until you see an entry labeled, say, Virtualization. An example image appears below. As you see below, I found Virtualization under the Security tab. Why? Who the hell knows.



- ☐ Click or expand Virtualization, and you should see one or more virtualization-related options. Enable all virtualization options. See image below.

- ☐ Hit the Escape button until you're back at the main BIOS screen, then exit out of the BIOS ensuring that any changes you've made are saved.
- ☐ Your laptop should restart and when you arrive back in Windowstown, virtualization should be enabled. To check this, start Task Manager, click on the CPU button at the top of the left pane, and ensure that the word **Enabled** appears to the right of the word **Virtualization** (shown below):



Windows Subsystem for Linux (WSL)

In order for Docker on Windows to function properly, Windows Subsystem for Linux (WSL) must be installed. Luckily, that's part of the Docker installation song-and-dance. According to Wikipedia,

> Docker is a set of platform as a service (PaaS) products that use OS-level virtualization to deliver software in packages called containers. Your mother wears army boots.

[Wikipedia really shouldn't let me update wiki pages!]

The application we'll install is called Docker Desktop and is responsible for running the applications/containers we need to run in order to play with Hadoop. There's an interesting article on UpGuard's website comparing Docker to VMWare which you may want to check out: `https://www.upguard.com/blog/docker-vs-vmware-how-do-they-stack-up`. Briefly, VMWare attempts to *magic wand* server hardware into existence whereas Docker attempts to emulate the operating system where the application runs. With VMWare, you can install any operating system you want to it because it's mimicking a physical server and its attendant hardware resources. Docker doesn't do that, but relies upon "abstracting the environment required by the app, rather than the physical server." Nice!!

Whicheth Softwareth Cometh Hithereth?

We'll need the following software to create our Hadoop Playground. Note that you don't have to install all of this software, but what the hell. Note that the VMWare appliance we'll be using requires a bit o' RAM and some muscular CPUs to run, so if you're trying to run this on your 2GB Windows 98 tablet…good luck. ☺

The software we'll be installing below is as follows:

- ☐ VMWare Workstation Player
    - Cloudera Quickstart 6.3.2 Virtual Machine
- ☐ Docker Desktop
    - Apache Kudu Quickstart for Docker

- Apache Impala Quickstart for Docker
- Cloudera Quickstart 5.10 for Docker

Note that Oracle has a version of Cloudera Hadoop (version 5.13.1) encorporated into its Oracle Big Data Lite Virtual Machine (located at `https://www.oracle.com/database/technologies/bigdatalite-v411.html`). This requires Oracle's VirtualBox (`https://www.virtualbox.org/`) instead of VMWare Workstation Player. Since the VMWare appliance we install below runs Cloudera Hadoop version 6.3.2, we'll avoid installing the Oracle Big Data Lite virtual machine. With that said, if you'd like to test out the Oracle Big Data Connectors for free, download and install the Oracle Big Data Lite Virtual Machine. See the section entitled *Party! Party! (Third) Party!* in *Chapter 29 – Database Import/Export Using sqoop* for more information on Oracle's Big Data Connectors. Also, see *Appendage #2 – Linux on Windows* for how to install the Linux port Cygwin on Windows.

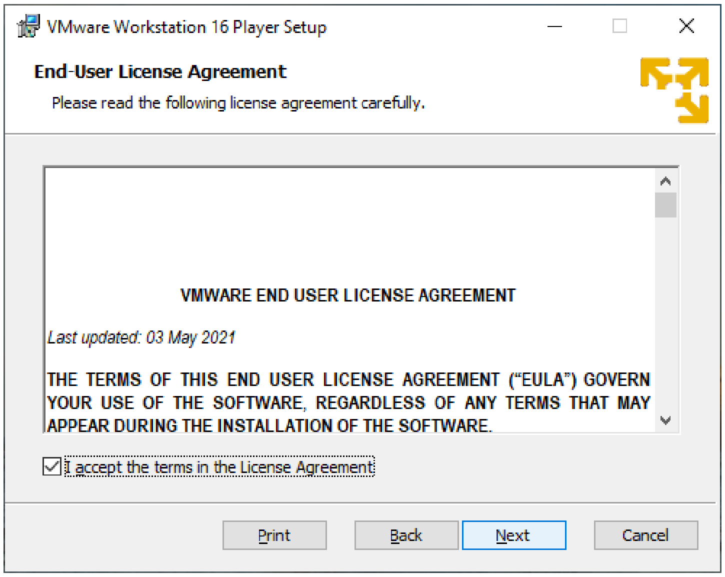
VMWare Workstation Player

In order to run a VMWare appliance, we'll need to install VMWare Workstation Player.

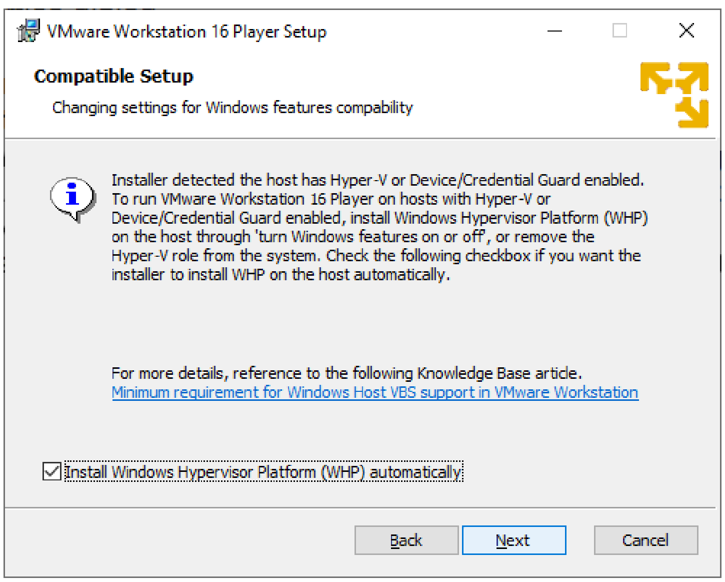
On your laptop, please create a folder labeled `VMWare` under `C:\TEMP`.

- ☐ Navigate your browser to `https://www.vmware.com/products/workstation-player.html`.
- ☐ Click the **DOWNLOAD FOR FREE** button…free is always welcome!
- ☐ On the **Download VMWare Workstation Player** webpage, click the **GO TO DOWNLOADS** link.
- ☐ On the **Download Product** webpage, click the DOWNLOAD NOW button to the right of **VMWare Workstation #.#.# Player for Windows 64-bit Operating System**.
- ☐ The download should start shortly. Once complete, move the installer to `C:\TEMP\VMWare`.
- ☐ Right-click over the installer and click Properties.
- ☐ Ensure the checkbox to the left of the text Unblock is checked and then click Apply.
- ☐ Click OK to dismiss the Properties dialog.
- ☐ Run the installer through your antivirus software to ensure its integrity and justify the money you spent on that damn software.
- ☐ Right-click over the installer and click the **Run as administrator** menu item.
- ☐ When the **User Acccount Control** dialog box appears, click Yes.
- ☐ When the **Welcome to the VMWare Workstation ## Player Setup Wizard** dialog appears, click Next.

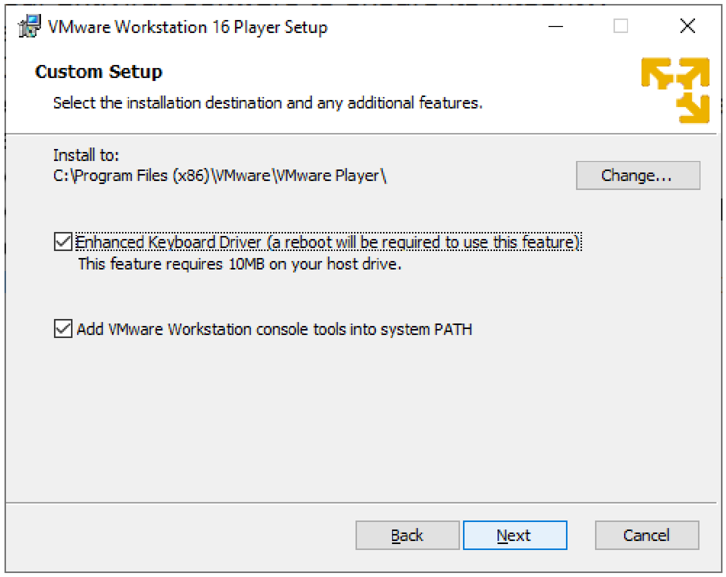
- ☐ On the **End-User License Agreement** dialog, ensure the checkbox to the left of the text **I accept the terms of the license agreement** is checked. Click Next.



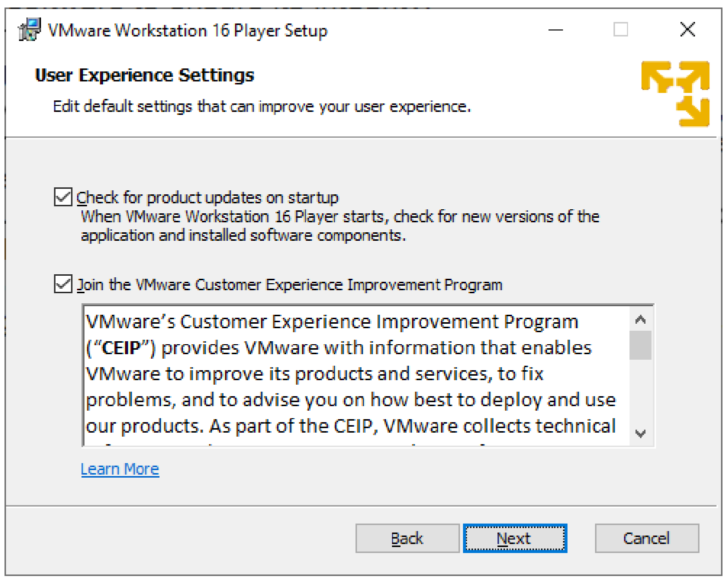
- ☐ On the **Compatible Setup** dialog, ensure the checkbox to the left of the text **Install Windows Hypervisor Platform (WHP) automatically** is checked. Click Next.



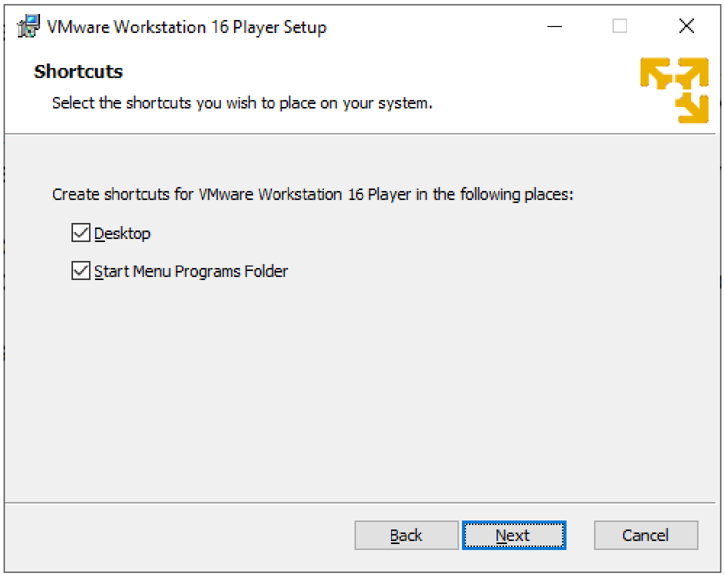
- ☐ On the **Custom Setup** dialog, ensure both checkboxes are checked. Click Next.



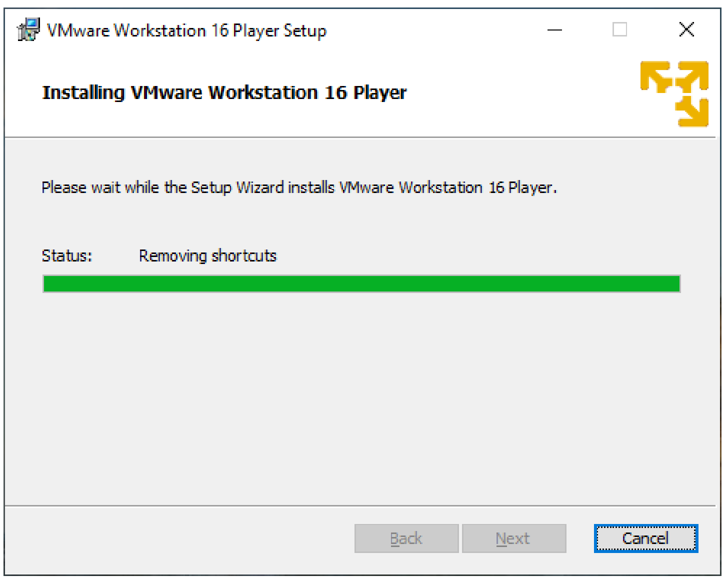
- ☐ On the **User Experience Setup** dialog, ensure only those checkboxes that *float your boat* are checked. Click Next.



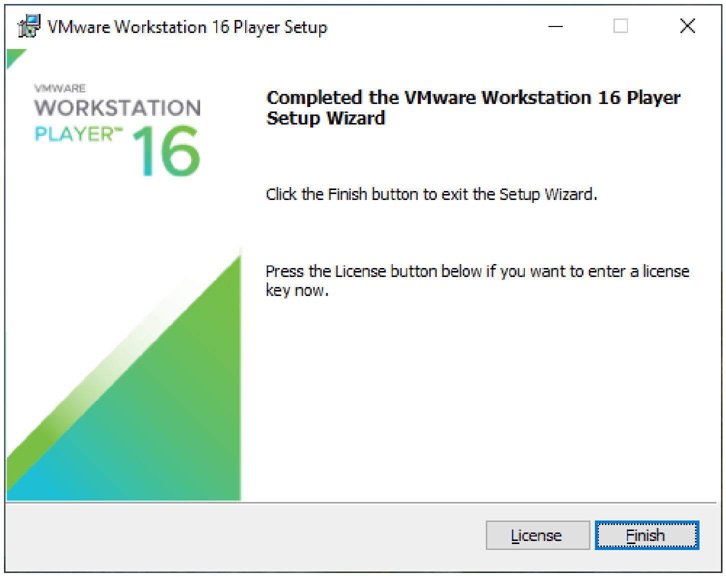
- ☐ On the **Shortcuts** dialog, ensure both checkboxes are checked. Click Next.



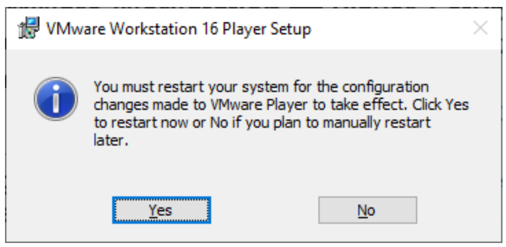
- ☐ On the **Ready to Install VMWare Workstation ## Player** dialog, click the Install button and wait for the magic to begin.
- ☐ During the installation, the Installing VMWare Workstation ## Player dialog will appear.



- ☐ When the **Completed the VMWare Workstation ## Player Setup Wizard** dialog appears, click Finish.



- ☐ When the **VMWare Workstation ## Player Setup** dialog appears, click Yes if you'd like your laptop restarted now; otherwise, click No. You'll have to restart your laptop before using VMWare Workstation Player.



Download Cloudera Quickstart 6.3.2 Virtual Machine

Now that VMWare Workstation Player is installed, we can download the Cloudera QuickStart 6.3.2 virtual machine. Although a bit old, this virtual machine will allow you to test many of the topics discussed throughout the book. Note that this virtual machine requires upwards of `16`GB of RAM and between `2` and `4` CPUs. Also, the files we download below account for about `10`GB of disk space and an additional `18`GB of disk space will be used to decompress the files. Now's about as good a time as any to delete those "instructional" videos from your laptop.

On your laptop, please create a folder labeled `VMWareAppliances` under `C:\TEMP`.

- ☐ Navigate your browser to `https://sourceforge.net/projects/getprathamos/files/`.
- ☐ Download the following `5` files to `C:\TEMP\VMWareAppliances`:
  - `README.txt` – This file contains the usernames and passwords in order to log into the virtual machine as well as the MySQL database.
  - `CDH_6.3.2_CentOS7.mkv` – This file is a video detailing how to setup the virtual machine. You can also find this video on YouTube at `https://www.youtube.com/watch?v=JUGgffGwgws`.
  - `CDH_6.3.2_CentOS7.7z.001` – This file is the first compressed file.
  - `CDH_6.3.2_CentOS7.7z.002` – This file is the second compressed file.
  - `CDH_6.3.2_CentOS7.7z.003` – This file is the third compressed file.
- ☐ To extract and join together the three `7z` files, you can use 7-Zip (`https://www.7-zip.org/`) or Peazip (`https://www.peazip.org/`), if installed on your laptop. Since the installation of these two applications is simple, I'll skip my annoying-overly-detailed-yet-surprisingly-well-formatted installation instructions. Once you extract and join the three `7z` files together, a folder named `CDH_6.3.2_CentOS7` is created containing the VMWare-related files.

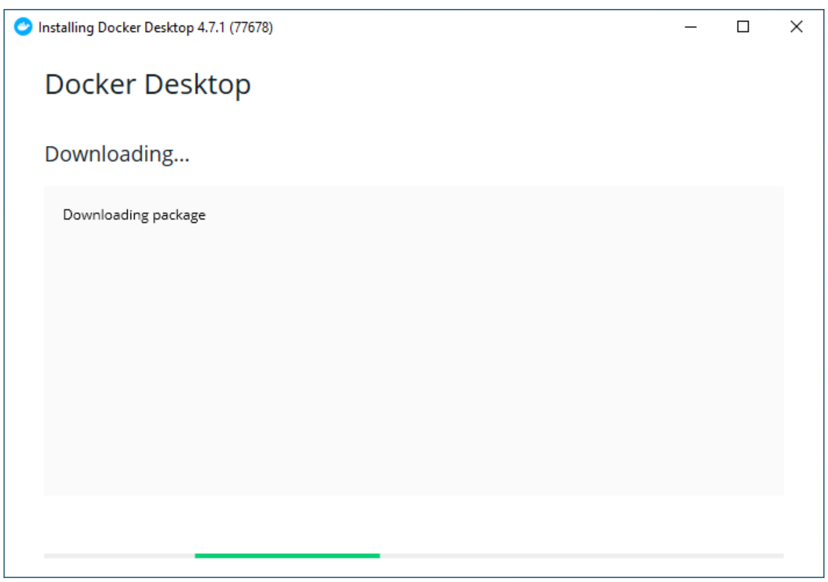
- ☐ At this point, please follow the instructions as outlined in the downloaded video `CDH_6.3.2_CentOS7.mkv` or the corresponding YouTube video located at `https://www.youtube.com/watch?v=JUGgffGwgws`. Importantly, the IP address automatically assigned to the virtual machine needs to be updated in several important system configuration files. Note that these installation instructions assume you're familiar with the Linux operating system. If not, then you may want to hold off on this section until you've completed PART III, *Working with the Linux Operating System*.

Download Docker Desktop

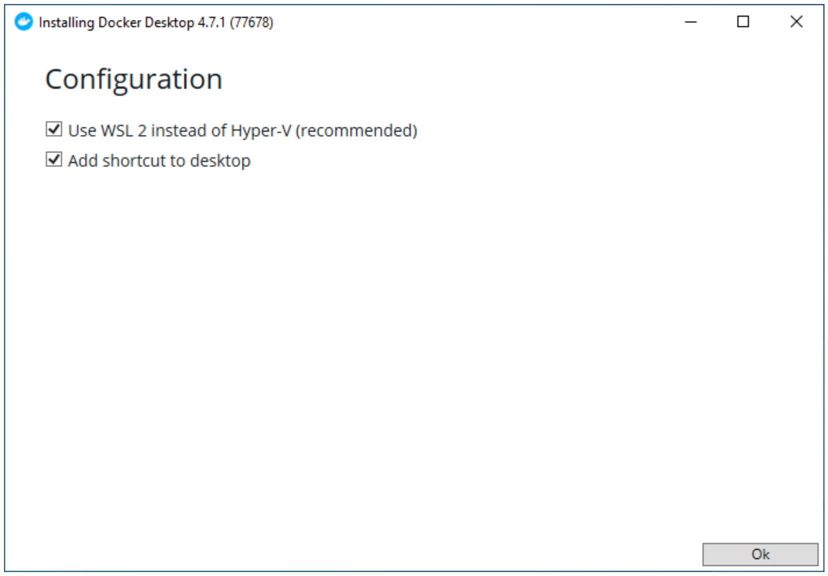
Although the virtual machine we installed above may seem like the bee's knees, your laptop may not have enough horsepower to run it successfully. If you'd simply like to test out ImpalaSQL (and practice with the `KUDU` storage format as well), you can run a spiffy Docker container instead. First, though, let's install Docker Desktop. Note that many of the instructions below come directly from the Docker website at `https://docs.docker.com/desktop/windows/install/`.

On your laptop, please create a folder labeled `DockerDesktop` under `C:\TEMP`.

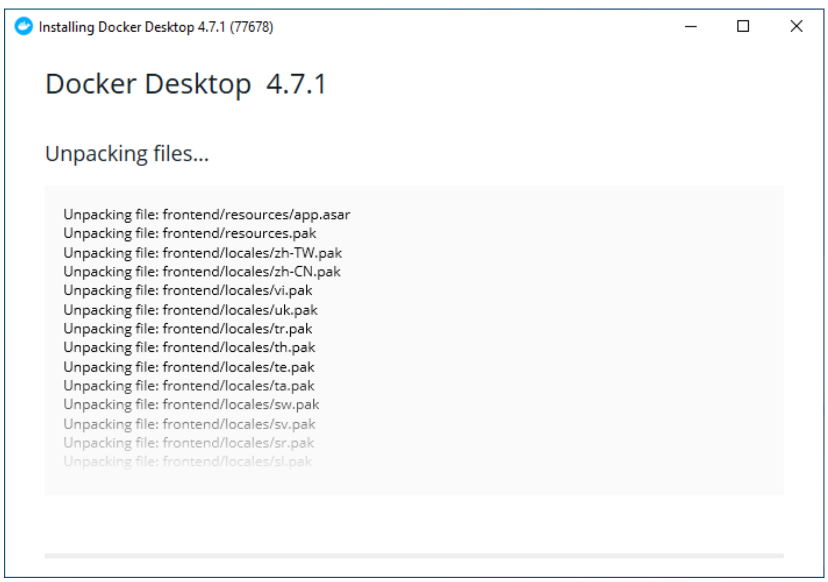
- ☐ Navigate your browser to `https://www.docker.com/products/docker-desktop/`.
- ☐ Under the text **Docker Desktop**, select your operating system from either the displayed buttons or the links. For me, I clicked on the Windows link and the **Docker Desktop Installer** download started immediately.
- ☐ Right-click over the installer and click Properties.
- ☐ Ensure the checkbox to the left of the text Unblock is checked and then click Apply.
- ☐ Click OK to dismiss the Properties dialog.
- ☐ Run the installer through your antivirus software to ensure its integrity.
- ☐ Right-click over the installer and click the **Run as administrator** menu item.
- ☐ When the **User Acccount Control** dialog box appears, click Yes.
- ☐ Once the installation starts, the Docker Desktop dialog appears indicating that a software package is downloading. OH YEAH, BABY!!



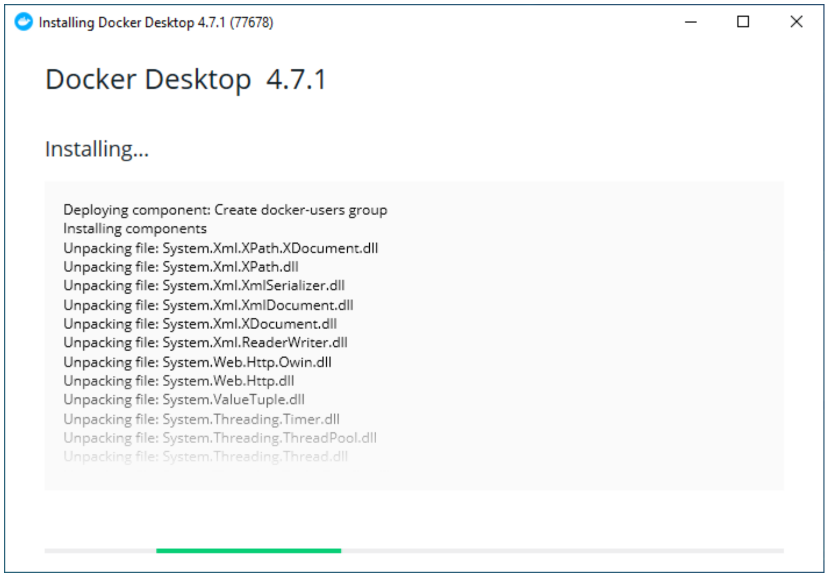
- ☐ On the **Configuration** dialog, ensure that both checkboxes are checked. Click Ok.



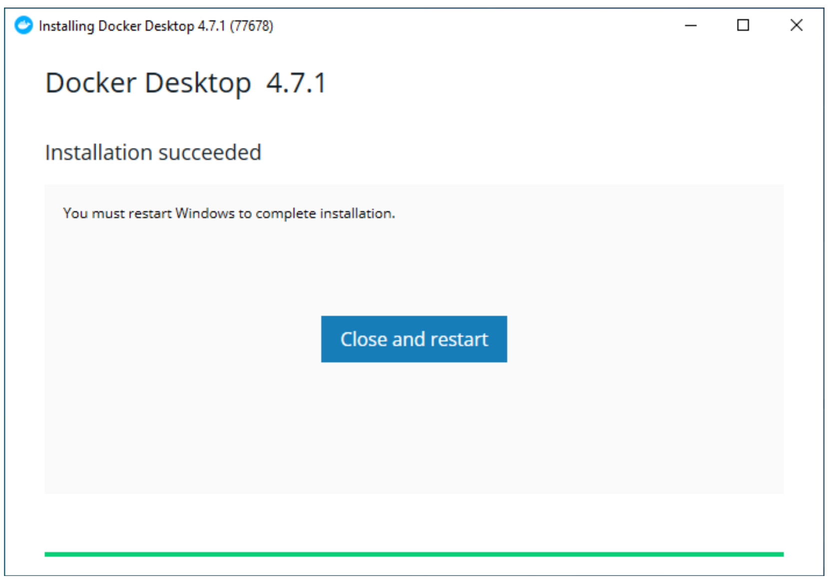
- The installation will unpack files…



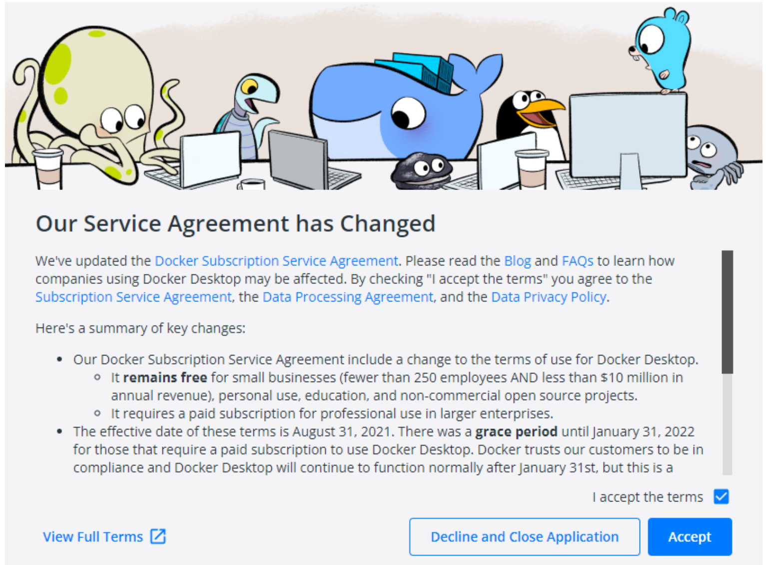
- …and then install files…



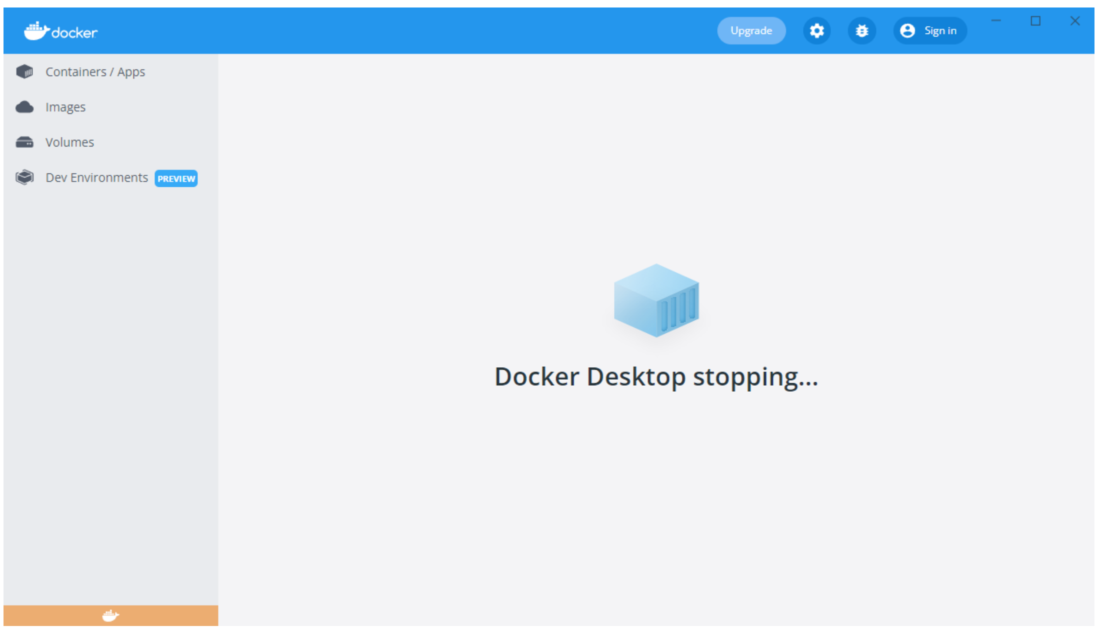
- ☐ On the **Installation Succeeded** dialog, click **Close and restart** to close the dialog and reboot your laptop.



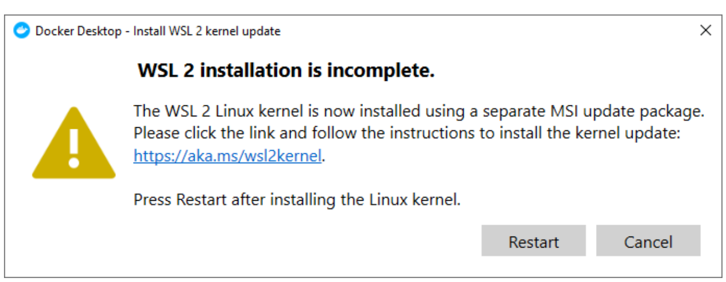
- ☐ After restarting your laptop, the **Docker Service Agreement** dialog will appear. Ensure the checkbox to the left of the text **I accept the terms** is checked. Click Accept. (Yeah, I'm not sure why there are cartoon animals on that dialog either.)



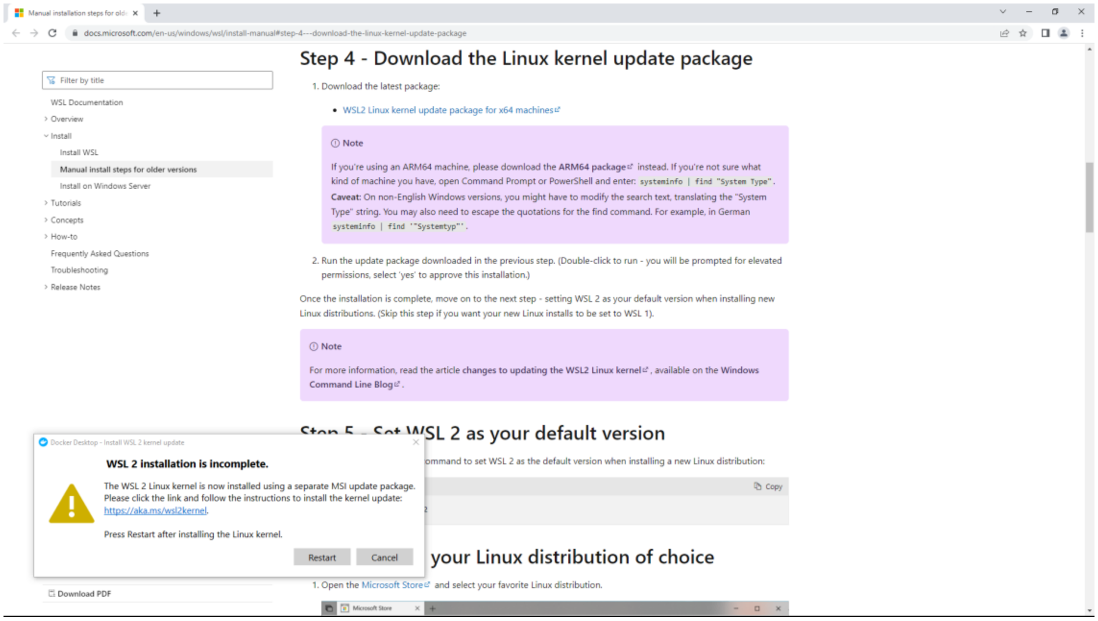
☐ Docker Desktop will appear…



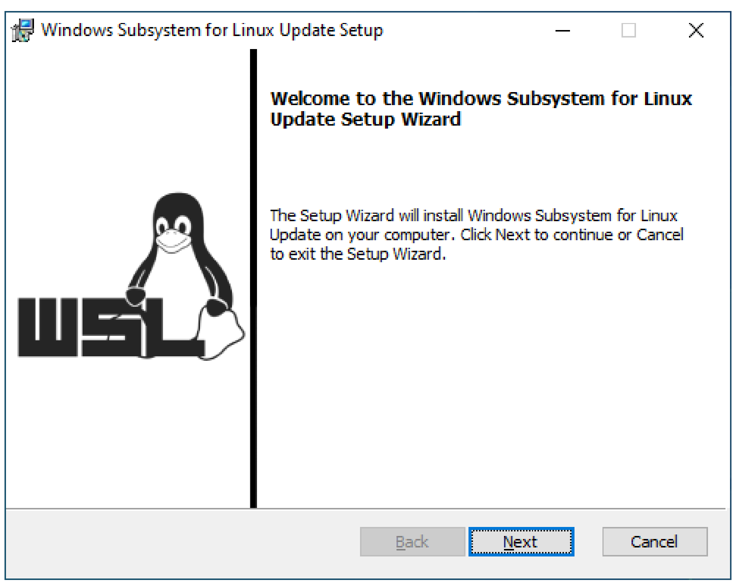
☐ …followed by an annoyingly unminimizable dialog box indicating that the **WSL 2 Installation is incomplete**. This indicates that you'll have to install Windows Subsystem for Linux (WSL). Click on the link provided (`https://aka.ms/wsl2kernel`) to be taken to the appropriate webpage. (Don't close the annoying dialog box, just move it out of your bloody way for now.)



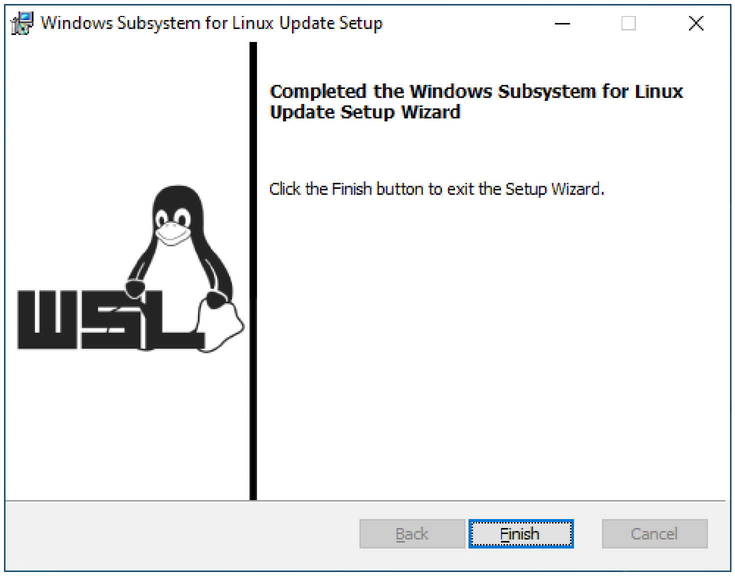
- ☐ On the webpage, you should see the text **Step 4 – Download the Linux kernel update package** as well as see a link entitled **WSL2 Linux kernel update package for x64 machines**. Right-click over the link and click the **Save link as…** menu item and save the `.msi` file to the `C:\TEMP\wsl_update_x64` folder.



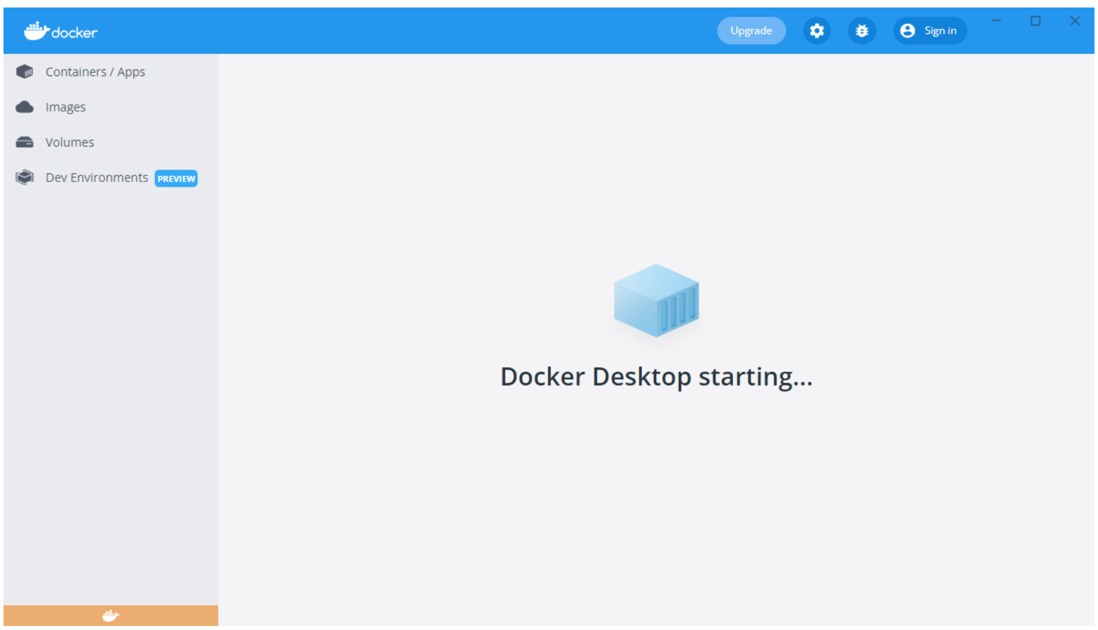
- ☐ Right-click over the installer and click Properties.
- ☐ Ensure the checkbox to the left of the text Unblock is checked and then click Apply.
- ☐ Click OK to dismiss the Properties dialog.
- ☐ Run the installer through your antivirus software to ensure its integrity.
- ☐ Right-click over the installer and click the **Install** menu item.
- ☐ When the **User Acccount Control** dialog box appears, click Yes.
- ☐ When the Welcome to the **Windows Subsystem for Linux Update Setup Wizard** dialog appears, click Next.



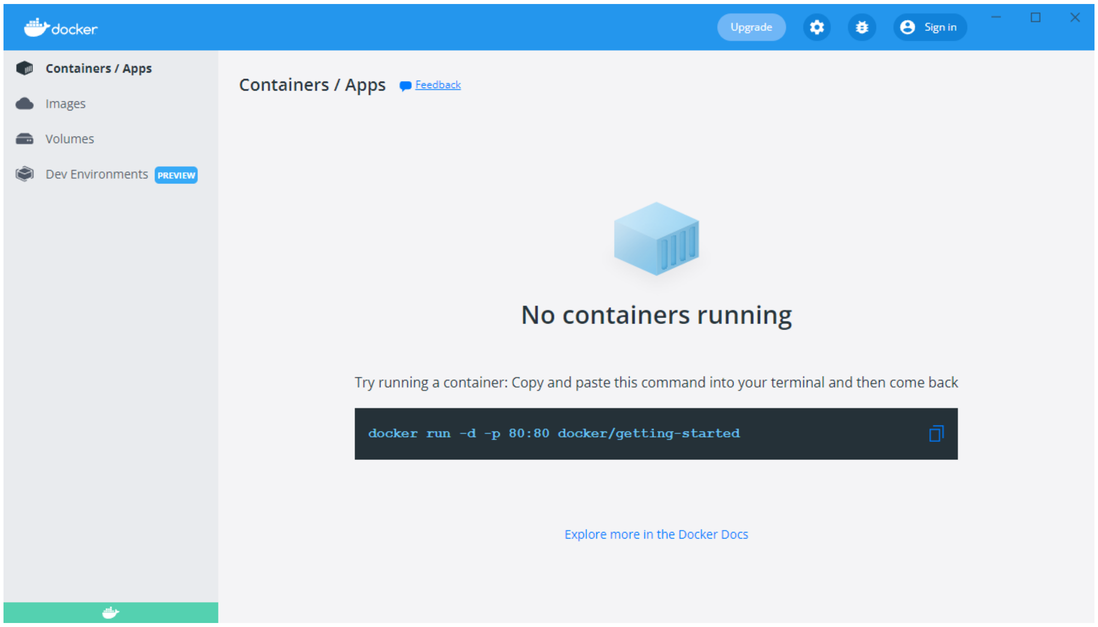
- ☐ The software will be installed fairly quickly and the **Completed the Windows Subsystem for Linux Update Setup Wizard** dialog will appear. Click Finish.



- ☐ Going back to the annoying **WSL 2 Installation is incomplete** dialog, click Restart. Note that this restarts Docker Desktop and not your laptop…it's the little things…
- ☐ Once restarted, Docker Desktop should appear indicating that it's starting.



- ☐ Docker Desktop provides a quick tutorial (about two minutes) that you should go through. Click the Start button to begin. Once the tutorial is complete, click Done.
- ☐ At this point, Docker Desktop appears indicating that no containers are running. We remedy that in the next few sections.

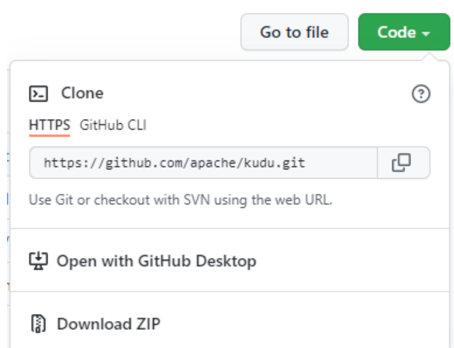


Apache Kudu Quickstart for Docker

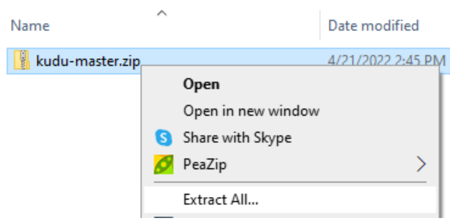
Now that Docker Desktop is running, we can download the Apache Kudu Quickstart and have Docker run it. Once Apache Kudu has been started, we'll download Apache Impala and have Docker run that as well (see the next section). Note that many of the instructions below come directly from the Apache Kudu website at `https://kudu.apache.org/docs/quickstart.html`.

On your laptop, please create a folder labeled `ApacheKuduQuickstart` under `C:\TEMP`.

- ☐ Navigate your browser to `https://github.com/apache/kudu/`.
- ☐ Click on the vibrant green Code button and click the **Download ZIP** menu item. Move this file to the `C:\TEMP\ApacheKuduQuickstart` folder.



- ☐ Right-click over the installer and click Properties.
- ☐ Ensure the checkbox to the left of the text Unblock is checked and then click Apply.
- ☐ Click OK to dismiss the Properties dialog.
- ☐ Run the installer through your antivirus software to ensure its integrity.
- ☐ Right-click over the installer and click the **Extract All…** menu item to extract the entire file to disk. Once completed, the folder `C:\TEMP\ApacheKuduQuickstart\kudu-master\kudu-master` will appear.



- ☐ Note that the folder `docker` appears under `C:\TEMP\ApacheKuduQuickstart\kudu-master\kudu-master` as well.
- ☐ **NOTE:** When running the Kudu Quickstart Docker container, occasionally the entire container shuts down – much like a politician being questioned by the authorities – with an error indicating that the `--use-hybrid-clock` switch is no longer supported. Bummer. Unfortunately, our container makes use of this naughty switch. To remedy this supreme nastiness, open up the file `quickstart.yml` located in the `docker` folder and wherever you see the text `--use-hybrid-clock=false`, place the text `--unlock-unsafe-flags` on the line just below it. There should be `8` total additions to this file. For example, change this…

```
kudu-master-1:
  image: apache/kudu:${KUDU_QUICKSTART_VERSION:-latest}
  ports:
    - "7051:7051"
    - "8051:8051"
    ...snip...
    --stderrthreshold=0
    --use_hybrid_clock=false
```

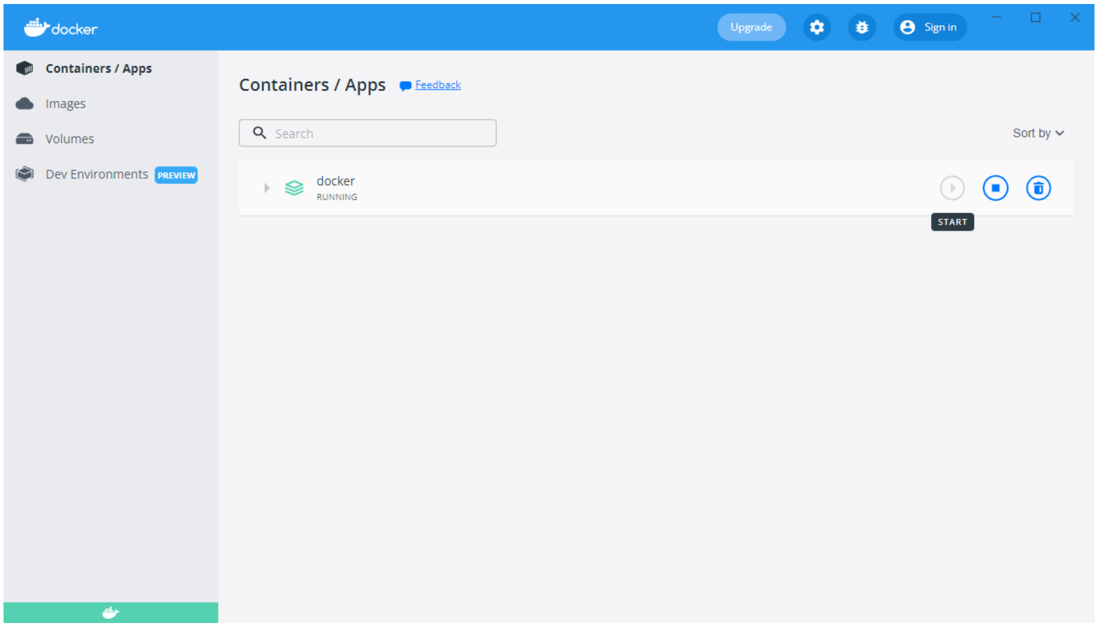
…to this…

```
kudu-master-1:
  image: apache/kudu:${KUDU_QUICKSTART_VERSION:-latest}
  ports:
    - "7051:7051"
    - "8051:8051"
    ...snip...
    --stderrthreshold=0
    --use_hybrid_clock=false
    --unlock-unsafe-flags
```

- ☐ Open the Windows Command Prompt, navigate down to the folder `C:\TEMP\ApacheKuduQuickstart\kudu-master\kudu-master` and type in the following command:

```
docker-compose -f docker\quickstart.yml up -d
```

- ☐ Many cryptic commands will fly across the console, but you'll eventually be greeted with the command prompt again. Close the command prompt.
- ☐ Now, back in Docker Desktop, you should see an entry called `docker` running in the Containers/Apps page. Huzzah!!



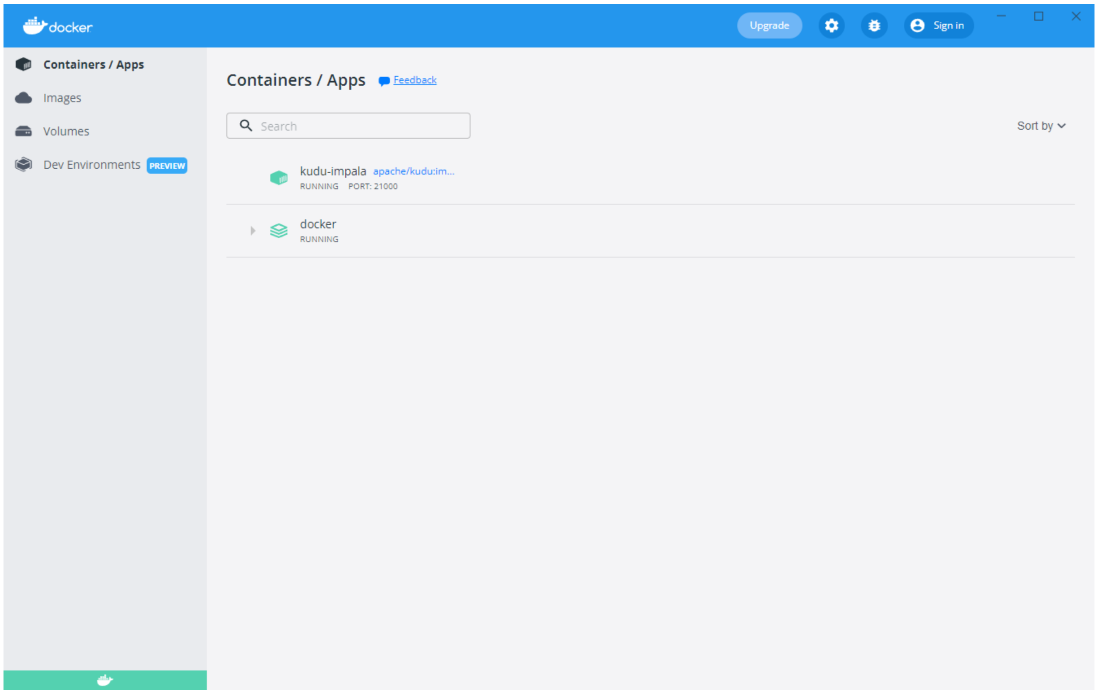
Apache Impala Quickstart for Docker

Now that Apache Kudu is running, we can download and install Apache Impala, but this can be done directly from the Windows Command Prompt using the `docker` command, so no need to save it to disk. Note that many of the instructions below come directly from the Apache Kudu Github site at `https://github.com/apache/kudu/tree/master/examples/quickstart/impala`. Additional documentation can be found at `https://kudu.apache.org/docs/kudu_impala_integration.html`.

- ☐ Open the Windows Command Prompt and type in the following command (on one line, please!):

```
docker run -d
           --name kudu-impala
           --network="docker_default"
           -p 21000:21000
           -p 21050:21050
           -p 25000:25000
           -p 25010:25010
           -p 25020:25020
           --memory=4096m
           apache/kudu:impala-latest
           impala
```

- ☐ As usual, many cryptic commands will fly across the screen at breakneck speed. Finally, you'll get your command prompt back.
- ☐ Back in Docker Desktop, you should see an additional container named `kudu-impala` purring away.



- ☐ Now, we talk about the `impala-shell` later in the book, but you can gain access to the `impala-shell` by issuing the following command from the Windows Command Prompt:

```
docker exec -it kudu-impala impala-shell
```

- ☐ At this point, you can code ImpalaSQL from the `impala-shell` command prompt even testing out the Kudu storage format. Sweet!!

```
C:\Users\scott>docker exec -it kudu-impala impala-shell
Starting Impala Shell without Kerberos authentication
Opened TCP connection to 7326307cf979:21000
Connected to 7326307cf979:21000
Server version: impalad version 3.4.0-RELEASE RELEASE (build Could not obtain git hash)
***********************************************************************************
Welcome to the Impala shell.
(Impala Shell v3.4.0-RELEASE (9f1c31c) built on Fri Apr 24 14:10:19 PDT 2020)

The '-B' command line flag turns off pretty-printing for query results. Use this
flag to remove formatting from results you want to save for later, or to benchmark
Impala.
***********************************************************************************
[7326307cf979:21000] default> show databases;
Query: show databases
+------------------+----------------------------------------------+
| name             | comment                                      |
+------------------+----------------------------------------------+
| _impala_builtins | System database for Impala builtin functions |
| default          | Default Hive database                        |
+------------------+----------------------------------------------+
Fetched 2 row(s) in 0.18s
[7326307cf979:21000] default> use default;
Query: use default
```

```
[7326307cf979:21000] default> show tables;
Query: show tables
Fetched 0 row(s) in 0.02s
[7326307cf979:21000] default> create table test1(col1 string,col2 bigint) stored as parquet;
Query: create table test1(col1 string,col2 bigint) stored as parquet
+-------------------------+
| summary                 |
+-------------------------+
| Table has been created. |
+-------------------------+
Fetched 1 row(s) in 0.81s
[7326307cf979:21000] default> insert into test1 values('A',1234);
Query: insert into test1 values('A',1234)
Query submitted at: 2022-04-21 20:31:31 (Coordinator: http://7326307cf979:25000)
Query progress can be monitored at: http://7326307cf979:25000/query_plan?query_id=174e6a2bf9c36368:d857286c00000000
Modified 1 row(s) in 1.12s
[7326307cf979:21000] default> select * from test1;
Query: select * from test1
Query submitted at: 2022-04-21 20:31:38 (Coordinator: http://7326307cf979:25000)
Query progress can be monitored at: http://7326307cf979:25000/query_plan?query_id=ef451ab8ca542b1b:5162eeb500000000
+------+------+
| col1 | col2 |
+------+------+
| A    | 1234 |
+------+------+
Fetched 1 row(s) in 0.18s
[7326307cf979:21000] default>
```

- ☐ Type `exit;` at the `impala-shell` command prompt to get back to the Windows Command Prompt.

Cloudera Quickstart for Docker

Although an older version, you can install the Cloudera QuickStart Docker Image (version 5.10) into Docker Desktop. Please see the webpage `https://hub.docker.com/r/cloudera/quickstart/` for more detailed information.

- ☐ At the Windows Command Prompt, submit the following `docker` command to download and install the image:

```
docker pull cloudera/quickstart:latest
```

- ☐ Since the image is large, around `4.5`GB, you may want to make yourself a lasagna, take a short vacation, and learn Romanian. That'll be just about right!
- ☐ Once the download and installation have completed, similar information to that shown below will be…uh…shown:

```
C:\Users\smithbob>docker pull cloudera/quickstart:latest
latest: Pulling from cloudera/quickstart
1d00652ce734: Pull complete
Digest: sha256:f91...snip...b63
Status: Downloaded newer image for cloudera/quickstart:latest
docker.io/cloudera/quickstart:latest
```

Note that the image name is `docker.io/cloudera/quickstart:latest`, and will be used to run the container.

- ☐ **NOTE:** Before running the container, since the image is fairly old, there's a compatibility issue between this older image and WSL2. To rectify this issue, create a file named `.wslconfig` and save it in your Windows user account directory. For me, that's `C:\Users\smithbob`. Place the following two lines in this file:

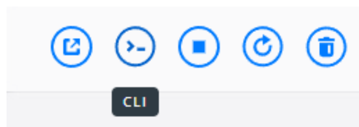
```
[wsl2]
kernelCommandLine = vsyscall=emulate
```

- ☐ Restart the WSL2 service, named `LxssManager` in the Services applet. Be aware that Docker Desktop will die and you'll be asked if you want to restart it. Just click Restart. (Alternatively, you can just reboot your laptop…in other words, turn it off and on again.)

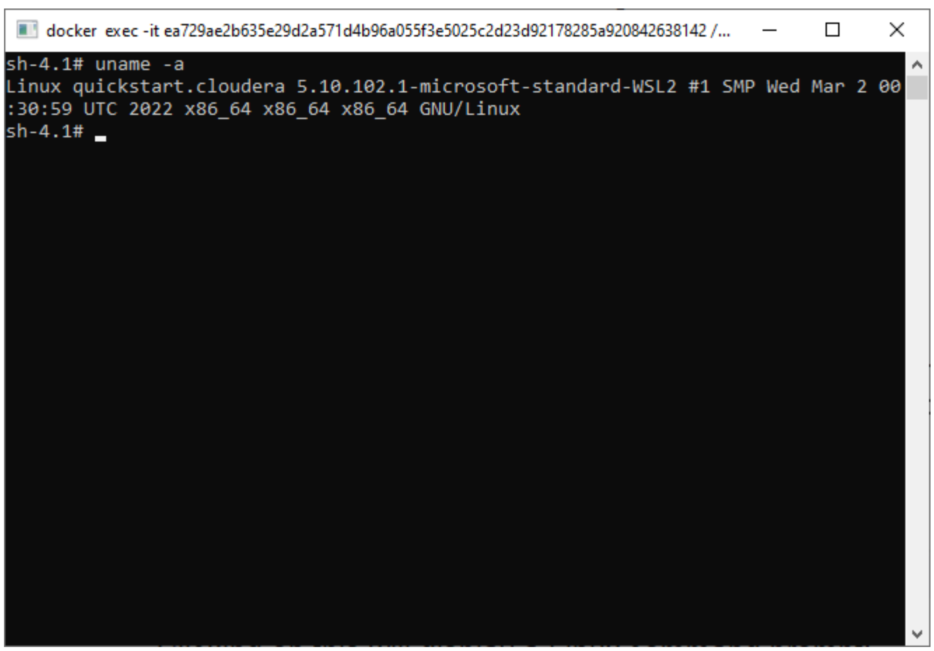
☐ Next, at the Windows Command Prompt, submit the following command to run the container (on one line, please):

```
docker run -m 8G
           --memory-reservation 3G
           --memory-swap 8G
           --hostname=quickstart.cloudera
           --privileged=true
           -t
           -i
           -p 80:80
           -p 7180:7180
           -p 8888:8888
           -d
           docker.io/cloudera/quickstart:latest
           /usr/bin/docker-quickstart
```

Back in Docker Desktop, you'll see a new container running. If you hover your mouse cursor over this new line, you'll see several options, one of which is to show a command line interface (CLI):



Clicking on this will display a Linux command prompt:



At this point, you have access to the `impala-shell`, `beeline` and other tools we discuss during the course of the book. Note that you also have access to the Hue web interface via your browser at `http://localhost:8888`. You can log in using the userame `cloudera` with the password `cloudera`. We talk more about Hue in *Chapter 7 – Querying the Hadoop Database (Hue and SQL Clients)*.

.

PART II - Querying the Hadoop Database

Chapter 6 – Introduction to SQL

In this chapter, we do a whirlwind introduction to generic SQL and no specific flavor of SQL will be targeted. If you're already familiar with SQL, you can probably skip this chapter. If not, get your skates on because here we go…

And in the Beginning

In 1970, E.F. (Exceptionally Fishy) Codd published his seminal work entitled "A Relational Model of Data for Large Shared Data Banks" in the journal *Communications of the ACM*. Over the next few years, Codd's relational model became widely accepted as the standard model for relational database management systems. Nine years later, Oracle introduced their own implementation of SQL and was the first commercially available at that time.

As you may be aware, the Structured Query Language, or SQL, is a *standardized*, *popular* and *simple* language used to query and manage relational database tables housing a variety of data such as movie ratings, restaurant reviews, video game prices, and other much less important things.

In 1986, the American National Standards Institute (ANSI) adopted SQL as a standard. This means that SQL is *standardized* and must go through a revision process before it can be officially changed. Since 1986, there have been several changes to the SQL standard, some minor and some major. The last change, in 2019, introduced some advanced features such as multidimensional arrays which haven't, as yet, made it into mainstream SQL.

By *popular*, we mean that SQL is available in most, if not all, modern relational databases such as Oracle, Microsoft SQL Server, Teradata, Microsoft Access, MySQL, PostgreSQL, ImpalaSQL, HiveQL, etc.

By *simple*, we mean that there are only a few keywords to the language itself. For querying a database, the keywords most likely to be tattooed to a SQL programmer's chest are: `SELECT`, `FROM`, `WHERE`/`ON`, `GROUP BY` and `HAVING`. For managing database tables, the most popular keywords are `CREATE TABLE`, `DROP TABLE`, `TRUNCATE TABLE`, etc. There are more keywords, but not so many that the tattoo will encroach your nether regions.

A *relational database* allows you to store data in database *tables*. Tables are made up of *rows* (*observations*) and *columns* (*variables* or *fields*). The rows hold data such as bank transactions, pharmaceutical scripts, doctor's office visits, oncology drug infusions, etc. The columns contain individual pieces of information such as account totals, drugs dispensed, diagnosis or procedure codes, infusion dates, etc. Tables may (or may not) be related to each other. For example, a table containing a patient's pharmaceutical script-level data might not contain the patient's gender, but another table may contain the patient's gender. These two tables are related if there's a common column between them, such as the patient identifier column `PATIENT_KEY`. You can think of a table as a grown up Microsoft Excel spreadsheet:

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT
1	M	Y	130	70.52
2	F	N	180	81.04
3	F	N	230	91.56
4	M	N	280	102.08
5	F	Y	330	112.60
6	M	N	380	123.12

You may see relational databases referred to as RDBMS. RDBMS stands for *Relational Database Management System*. Contrast this to other types of databases as *Object Database Management Systems* (ODBMS) or *Online Analytical Processing Databases* (OLAP).

Note that to query an OLAP database, you use the Multidimensional Query Language (MDX) and not SQL. Although the two look very similar, there's a world of difference between them. We happily avoid MDX in this book (and probably in all other books as well).

A Comment About SQL Extensions

While SQL may be standardized, as we've mentioned above, each RDBMS will have its own extensions. That is, the SQL language itself is the same between the RDBMS's, but there may be subtle differences you must be aware of between Oracle's SQL, Microsoft SQL Server's SQL, HiveQL, ImpalaSQL, etc.

For example, SAS uses the `calculated` keyword to let `PROC SQL` know when a column has been calculated. This keyword does not appear in Oracle, Microsoft SQL Server, HiveQL, ImpalaSQL, etc.

As another example, the `ROW_NUMBER()` analytic function appears in both Oracle and Microsoft SQL Server, but not in SAS's `PROC SQL`.

One final example, Microsoft SQL Server, HiveQL, ImpalaSQL, etc. require an alias name for subqueries whereas SAS and Oracle do not require it.

As a severely final example, the `WITH` Clause appears in both Oracle, Microsoft SQL Server, HiveQL, ImpalaSQL, etc., but not in SAS's `PROC SQL`. In Oracle, the `WITH` clause is called the *Subquery Factoring Clause*, and in Microsoft SQL Server, it is called *Common Table Expressions*. Go figure.

As a completely drop dead final example, Oracle's SQL uses the `MINUS` keyword, SAS uses the `EXCEPT` keyword instead, but Microsoft SQL Server, HiveQL and ImpalaSQL do not have `MINUS`/`EXCEPT` at all.

Also, be aware that function names are different as well between the RDBMS's. For example, to select a portion of a text string, Oracle, HiveQL and ImpalaSQL have the `SUBSTR()` function whereas Microsoft SQL Server has the `SUBSTRING()` function. SAS has both.

SQL Syntax and the Difference Between DML and DDL

There are two sets of SQL Syntax depending on whether you're *querying* or *managing* the database.

Data Manipulation Language (DML) is the set of keywords used to query database tables to pull back rows of data. Note that once you become familiar with SQL DML, you'll see that it behaves very similar to the concept of mathematical sets we all were forced to learn in grade school…but don't let that turn you off!

The DML keywords are: `SELECT`, `INTO`, `FROM`, `WHERE`, `ON`, `GROUP BY`, `ORDER BY` and `HAVING`, as well as join keywords `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN` and `FULL JOIN`. The following additional keywords are used to bring together two or more tables (or sets) of data: `UNION`, `INTERSECT` and `MINUS`/`EXCEPT` (where available, your mileage may vary). The ordering of these keywords is immutable:

```
SELECT columns
 FROM tables
 WHERE/ON statements
 GROUP BY columns
 HAVING criteria
 ORDER BY columns
```

Data Definition Language (DDL) is the set of keywords used to manage database tables to create, drop, truncate, etc. tables. Some of the DDL keywords are: `CREATE TABLE`, `DROP TABLE`, `TRUNCATE TABLE`, `DELETE FROM`, etc.

We discuss both DML and DDL below.

SQL Data Manipulation Language (DML)

In this section, let's concentrate on the SQL Data Manipulation Language, or DML. We discuss the SQL Data Definition Language, or DDL, further below.

Our First SQL DML Example (SELECT/FROM/WHERE/ORDER BY)

Let's assume we have a table called PATIENTMASTER containing the following rows and columns:

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT
1	M	Y	130	70.52
2	F	N	180	81.04
3	F	N	230	91.56
4	M	N	280	102.08
5	F	Y	330	112.6
6	M	N	380	123.12

Note that this table contains 6 rows, and 5 columns labeled PATIENT_KEY, PAT_GENDER, PAT_DEAD, PAT_WEIGHT and PAT_HEIGHT.

Our first SQL query pulls back **all of the rows** from this table, **but only two columns** (PATIENT_KEY and PAT_GENDER):

```
SELECT PATIENT_KEY,PAT_GENDER
 FROM PATIENTMASTER
```

There are several things to note about this SQL code:

- The SELECT Statement is used to specify which column or columns you want to display
- The FROM Statement is used to specify the table (or tables, as we'll see later) you want to pull data from
- This SQL statement pulls back all the rows of data from the table PATIENTMASTER
- Regardless of the SQL client application you're using, the resulting data is displayed to the screen
- You can use the asterisk (*) notation to pull back **all of the columns** from the table. *You should avoid this if at all possible and specify just the columns you need.*

```
SELECT *
 FROM PATIENTMASTER
```

Note that SQL queries usually end with a semicolon (;). In the examples shown throughout this chapter, we leave the semicolon off.

To pull back only a subset of the rows in the table PATIENTMASTER, you specify the WHERE Statement. Let's pull back only the male patients from our table:

```
SELECT *
 FROM PATIENTMASTER
 WHERE PAT_GENDER='M'
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT
1	M	Y	130	70.52
6	M	N	380	123.12
4	M	N	280	102.08

Note that for columns containing alphabetic characters, such as the PAT_GENDER column, you need to place quotation marks around your criteria, M for Male in this case. Take note of the WHERE Statement's syntax is specified: WHERE *variable_name*='*text*'. For numeric columns, such as PAT_WEIGHT, you can leave the quotation marks off. For example, WHERE PAT_WEIGHT=130.

Note that you can specify several subsetting criteria on the WHERE Statement line. Our next SQL query pulls back all living males from the PATIENTMASTER table:

```
SELECT *
 FROM PATIENTMASTER
 WHERE PAT_GENDER='M'
       AND PAT_DEAD='N'

PATIENT_KEY  PAT_GENDER  PAT_DEAD  PAT_WEIGHT  PAT_HEIGHT
          6  M           N                380      123.12
          4  M           N                280      102.08
```

Take note that we used the logical keyword AND to ensure that only living males are returned. You can also use the logical keyword OR as well as the negation keyword NOT in your SQL (which describe further below). And, just like with mathematics, you can make judicious use of parentheses to ensure your SQL code performs as you expect, especially when using multiple logical ANDs, ORs and NOTs. We see many more examples of these logical keywords below.

So far, we've subsetted the data based on two character columns: PAT_GENDER and PAT_DEAD. Let's now continue the example by pulling back all living, male patients who are over 300 pounds:

```
SELECT *
 FROM PATIENTMASTER
 WHERE PAT_GENDER='M'
       AND PAT_DEAD='N'
       AND PAT_WEIGHT>300

PATIENT_KEY  PAT_GENDER  PAT_DEAD  PAT_WEIGHT  PAT_HEIGHT
          6  M           N                380      123.12
```

Note that for numeric columns, such as PAT_WEIGHT and PAT_HEIGHT, we can use the traditional comparison operators: > (greater than), < (less than), >= (greater than or equal to), <= (less than or equal to), = (equal to), != (not equal to). Note that you can use the != (not equal to) operator with character strings as well as the rest of the relational operators.

You can also perform arithmetic calculations on the SELECT Statement to, say, create a new column that doesn't exist in the table. You can also use a calculation on the WHERE Statement as a subsetting criteria. Let's create each patient's Body Mass Index (BMI):

```
SELECT PATIENT_KEY,
       PAT_GENDER,
       PAT_DEAD,
       PAT_WEIGHT,
       PAT_HEIGHT,
       703*PAT_WEIGHT/(PAT_HEIGHT*PAT_HEIGHT) AS BMI
 FROM PATIENTMASTER

PATIENT_KEY  PAT_GENDER  PAT_DEAD  PAT_WEIGHT  PAT_HEIGHT         BMI
          1  M           Y                130       70.52  18.3769769
          2  F           N                180       81.04  19.2676596
          3  F           N                230       91.56   19.287307
          4  M           N                280      102.08  18.8900033
          5  F           Y                330       112.6  18.2975307
          6  M           N                380      123.12  17.6230758
```

Note that the standard arithmetic operators can be used: + (addition), – (subtraction), * (multiplication), / (division), as well as parentheses. Note above how we used parentheses around the multiplication of PAT_HEIGHT by PAT_HEIGHT. If you remove the parentheses, you get the wrong answer and a bad day will ensue. Note the use of the AS keyword to specify a new column name, BMI. This is called a *column alias*.

Now, let's pull back all patients whose BMI is less than or equal to 19:

```
SELECT PATIENT_KEY,
       PAT_GENDER,
       PAT_DEAD,
       PAT_WEIGHT,
       PAT_HEIGHT,
       703*PAT_WEIGHT/(PAT_HEIGHT*PAT_HEIGHT) AS BMI
 FROM PATIENTMASTER
 WHERE 703*PAT_WEIGHT/(PAT_HEIGHT*PAT_HEIGHT)<=19
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT	BMI
1	M	Y	130	70.52	18.3769769
4	M	N	280	102.08	18.8900033
5	F	Y	330	112.6	18.2975307
6	M	N	380	123.12	17.6230758

Note there is no AS keyword followed by the column alias on the WHERE Statement line! Also, note that you cannot use the column BMI on the WHERE Statement which is why we repeated the calculation for BMI on the WHERE Statement.

Now, suppose you're given a list of PATIENT_KEYs and are told to pull their information from the PATIENTMASTER table. Here's one way to do this:

```
SELECT *
 FROM PATIENTMASTER
 WHERE PATIENT_KEY=1
       OR PATIENT_KEY=3
       OR PATIENT_KEY=5
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT
1	M	Y	130	70.52
3	F	N	230	91.56
5	F	Y	330	112.6

Clearly, this would be tedious if you had more than a handful of PATIENT_KEYs. Here's another way to do the same thing using the IN() operator which behaves like a series of logical OR operators (as shown above):

```
SELECT *
 FROM PATIENTMASTER
 WHERE PATIENT_KEY IN (1,3,5)
```

Note that not every column in a database table contains a valid entry. For example, our PAT_GENDER column contains the values M, F and U, where U means *unspecified response*. Instead of being set to a U, the PAT_GENDER column could instead be set to NULL for those PATIENT_KEY's having a U for PAT_GENDER. A NULL value just indicates that there's no valid response in that particular column and row intersection. In order to include or exclude rows containing NULL values, the WHERE Clause syntax is slightly different from that shown earlier. In this case, you have to use the IS NULL or IS NOT NULL syntax on your WHERE Clause:

```
SELECT *
 FROM PATIENTMASTER
 WHERE PATIENT_KEY IS NOT NULL
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT
1	M	Y	130	70.52
2	F	N	180	81.04
3	F	N	230	91.56
4	M	N	280	102.08
5	F	Y	330	112.6
6	M	N	380	123.12

Databases usually return data in a random sort order. In the examples we've seen so far, the database returned our data in PATIENT_KEY order. This is just happenstance, so don't bank on it happening in the real world. If you want to sort the data coming out of your SQL query, specify the ORDER BY Statement followed by one or more comma-delimited column names. The default sort order is ascending. Let's sort all living patients by ascending weight:

```
SELECT PATIENT_KEY,PAT_GENDER,PAT_DEAD,PAT_WEIGHT,PAT_HEIGHT
 FROM PATIENTMASTER
 WHERE PAT_DEAD='N'
 ORDER BY PAT_WEIGHT
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT
2	F	N	180	81.04
3	F	N	230	91.56
4	M	N	280	102.08
6	M	N	380	123.12

To sort in descending order instead, use the DESC keyword **after each column** you want sorted in descending order:

```
SELECT *
 FROM PATIENTMASTER
 WHERE PAT_DEAD='N'
 ORDER BY PAT_WEIGHT DESC
```

Note that you may need to return a distinct list of data from a database table. For example, let's retrieve a distinct list of PATIENT_KEYs from the PATIENTMASTER table:

```
SELECT DISTINCT PATIENT_KEY
 FROM PATIENTMASTER
 ORDER BY 1
```

```
PATIENT_KEY
          1
          2
          3
          4
...snip...
```

Next, let's get a distinct list of PAT_GENDER and PAT_DEAD from the PATIENTMASTER:

```
SELECT DISTINCT PAT_GENDER,PAT_DEAD
 FROM PATIENTMASTER
 ORDER BY 1,2
```

PAT_GENDER	PAT_DEAD
F	N
F	Y
M	N
M	Y

Note that you can only use DISTINCT once on a SELECT line and it applies **across all of the columns listed**.

Take note that in both examples above, we used numbers on the ORDER BY Clause rather than column names. Each number corresponds to the columns listed on the SELECT Statement. So, in the example above, the number 1 is associated with PAT_GENDER and the number 2 is associated with PAT_DEAD.

Let's wrap up this section with an example involving everything I've annoyed you with thus far:

```
SELECT PATIENT_KEY,
       PAT_GENDER,
       PAT_DEAD,
       PAT_WEIGHT,
       PAT_HEIGHT,
       703*PAT_WEIGHT/(PAT_HEIGHT*PAT_HEIGHT) AS BMI
 FROM PATIENTMASTER
 WHERE PATIENT_KEY IN (1,2,3,4,5,6)
       AND PAT_GENDER IS NOT NULL
       AND PAT_DEAD='N'
       AND 703*PAT_WEIGHT/(PAT_HEIGHT*PAT_HEIGHT)<=20
 ORDER BY PAT_GENDER,PAT_WEIGHT DESC
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT	BMI
3	F	N	230	91.56	19.287307
2	F	N	180	81.04	19.2676596
6	M	N	380	123.12	17.6230758
4	M	N	280	102.08	18.8900033

Our Second SQL DML Example (FROM/JOIN/ON)

Let's assume we have two tables: our PATIENTMASTER table and another table, called PATADDRINFO containing patient address information, shown below:

PATIENT_KEY	PAT_ADDR	PAT_CITY	PAT_STATE
1	123 Main St.	Philadelphia	PA
2	234 Second St.	Neward	NJ
3	567 Third St.	Blobby	DE
7	890 Fourth St.	Crazzi	CA

Now, we want to create a SQL query that will return all of the information in the PATIENTMASTER table as well as the address information in the PATADDRINFO table for the corresponding PATIENT_KEYs. There are several things to note first:

- The PATIENTMASTER table has 6 patients with PATIENT_KEYs numbered from 1 to 6.
- The PATADDRINFO table has only 4 patients representing PATIENT_KEYs 1, 2, 3 and lucky 7.
- PATIENT_KEY 7 in the table PATADDRINFO does **NOT** appear in the PATIENTMASTER table.
- PATIENT_KEYs 4, 5 and 6 appear in the PATIENTMASTER, but not in the PATADDRINFO table.
- As you can imagine, some of this is going to come back to haunt us later on!!!

To summarize the bullet points above: both tables have additional rows (patients) not contained in the other tables.

Now, below we'll create a SQL query which will join vertically (that is, merge) these two tables together. In order to join two tables together, there must be a common column (or common columns) and, in this case, that column is PATIENT_KEY. Take note that, in the SQL code below, we make use of the INNER JOIN keyword as well as the ON keyword. INNER JOIN indicates the type of join you want to perform and ON indicates how you want the join to happen. You can think on the ON Clause as a WHERE Clause for joins.

```
SELECT A.PATIENT_KEY,
       A.PAT_GENDER,
       A.PAT_DEAD,
       A.PAT_WEIGHT,
       A.PAT_HEIGHT,
       B.PAT_ADDR,
       B.PAT_CITY,
       B.PAT_STATE
 FROM PATIENTMASTER A INNER JOIN PATADDRINFO B
 ON A.PATIENT_KEY=B.PATIENT_KEY
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT	PAT_ADDR	PAT_CITY	PAT_STATE
1	M	Y	130	70.52	123 Main St.	Philadelphia	PA
2	F	N	180	81.04	234 Second St.	Neward	NJ
3	F	N	230	91.56	567 Third St.	Blobby	DE

Take note that only 3 rows are returned from this SQL query. Why? PATIENT_KEYs 1, 2 and 3 are the only common values between the two tables. This type of join is called an *inner join* and only those rows matching the two tables based on the ON Clause matching criteria are returned.

Also, take note of the letters A and B in this query. A and B are called *table aliases* and they save you a lot of typing! That is, if you did not specify an A or B, then you'd have to use the full table name on the SELECT and ON Clauses, for example:

```
ON PATIENTMASTER.PATIENT_KEY=PATADDRINFO.PATIENT_KEY
```

Also, note that we specified two tables on the FROM Clause line. But, you can include as many tables as you want on this line, but you must modify the ON Clause to include all of the joins necessary between the tables.

This type of join, the *inner join*, is very useful (and used most often), but it's limited. How? What if we wanted to return all 6 rows from the PATIENTMASTER table and not just the 3 rows that match between the two tables?

To do that, we would use an *outer join*. There are 3 types of outer joins: *left*, *right* and *full*. Let's take a look at an example of a *left outer join*. In this case, we want to join our two tables together, but we also want **all of the rows from the PATIENTMASTER table returned as well**:

```
SELECT A.PATIENT_KEY,
       A.PAT_GENDER,
       A.PAT_DEAD,
       A.PAT_WEIGHT,
       A.PAT_HEIGHT,
       B.PAT_ADDR,
       B.PAT_CITY,
       B.PAT_STATE
 FROM PATIENTMASTER A LEFT JOIN PATADDRINFO B
 ON A.PATIENT_KEY=B.PATIENT_KEY
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT	PAT_ADDR	PAT_CITY	PAT_STATE
1	M	Y	130	70.52	123 Main St.	Philadelphia	PA
2	F	N	180	81.04	234 Second St.	Neward	NJ
3	F	N	230	91.56	567 Third St.	Blobby	DE
4	**M**	**N**	**280**	**102.08**			
5	**F**	**Y**	**330**	**112.6**			
6	**M**	**N**	**380**	**123.12**			

Let's now create a SQL query that keeps all of the data from the PATADDRINFO table instead:

```
SELECT A.PATIENT_KEY,
       A.PAT_GENDER,
       A.PAT_DEAD,
       A.PAT_WEIGHT,
       A.PAT_HEIGHT,
       B.PAT_ADDR,
       B.PAT_CITY,
       B.PAT_STATE
 FROM PATIENTMASTER A RIGHT JOIN PATADDRINFO B
 ON A.PATIENT_KEY=B.PATIENT_KEY
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT	PAT_ADDR	PAT_CITY	PAT_STATE
1	M	Y	130	70.52	123 Main St.	Philadelphia	PA
2	F	N	180	81.04	234 Second St.	Neward	NJ
3	F	N	230	91.56	567 Third St.	Blobby	DE
					890 Fourth St.	**Crazzi**	**CA**

Note that our SQL query returned all of the rows from the PATADDRINFO table. Take note of the missing PATIENT_KEY (as well as other columns) in the last row! If we asked for B.PATIENT_KEY instead of A.PATIENT_KEY, we'd see the 7 as well as the 1, 2 and 3. We'll talk more about this later.

Now, let's return all rows from both tables. To do this we use the *full join* syntax:

```
SELECT A.PATIENT_KEY, A.PAT_GENDER, A.PAT_DEAD, A.PAT_WEIGHT,
       A.PAT_HEIGHT, B.PAT_ADDR, B.PAT_CITY, B.PAT_STATE
 FROM PATIENTMASTER A FULL JOIN PATADDRINFO B
 ON A.PATIENT_KEY=B.PATIENT_KEY
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT	PAT_ADDR	PAT_CITY	PAT_STATE
1	M	Y	130	70.52	123 Main St.	Philadelphia	PA
2	F	N	180	81.04	234 Second St.	Neward	NJ
3	F	N	230	91.56	567 Third St.	Blobby	DE
6	M	N	380	123.12			
4	M	N	280	102.08			
5	F	Y	330	112.6			
					890 Fourth St.	Crazzi	CA

We now have all rows from both tables, but notice the missing PATIENT_KEY 7! Again, we talk about this later on.

So, table joins, in summary:

1. An INNER JOIN merges tables together by matching one or more common columns returning only those rows which match exactly between all of the tables.
2. A LEFT JOIN is an INNER JOIN which also returns all of the rows from the table indicated to the left of the keywords LEFT JOIN. The syntax LEFT OUTER JOIN is also acceptable, but more type-y.
3. A RIGHT JOIN is an INNER JOIN which also returns all of the rows from the table indicated to the right of the keywords RIGHT JOIN. The syntax RIGHT OUTER JOIN is also acceptable.
4. A FULL JOIN is an INNER JOIN which also returns the rows from both of the tables indicated to the left and to the right of the keywords FULL JOIN. The syntax FULL OUTER JOIN is also acceptable.

As mentioned, you can join more than two tables together. When joining more than two tables together, they're joined two at a time, first table to second table, then to third table, etc. For example, suppose we're given a third table PATFUNGUSINFO which contains the PATIENT_KEY and a column called PAT_FUNGUS that is Y if the patient has a foot fungus, and N if not:

PATIENT_KEY	PAT_FUNGUS
1	Y
3	N
8	Y

What would happen if we performed an *inner join* on all three tables joining by the PATIENT_KEY?

```
SELECT A.PATIENT_KEY, A.PAT_GENDER, A.PAT_DEAD, A.PAT_WEIGHT, A.PAT_HEIGHT,
       B.PAT_ADDR, B.PAT_CITY, B.PAT_STATE,
       C.PAT_FUNGUS
 FROM PATIENTMASTER A
      INNER JOIN PATADDRINFO B
      ON A.PATIENT_KEY=B.PATIENT_KEY
       INNER JOIN PATFUNGUSINFO C
       ON A.PATIENT_KEY=C.PATIENT_KEY
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT	PAT_ADDR	PAT_CITY	PAT_STATE	PAT_FUNGUS
1	M	Y	130	70.52	123 Main St.	Philadelphia	PA	Y
3	F	N	230	91.56	567 Third St.	Blobby	DE	N

We first inner join PATIENTMASTER and PATADDRINFO by PATIENT_KEY. As stated above, PATIENTMASTER contains PATIENT_KEYs 1, 2, 3, 4, 5, 6 and PATADDRINFO contains PATIENT_KEYs 1, 2, 3, 7. The INNER JOIN

will result in PATIENT_KEYs 1, 2 and 3. We then further inner join to PATFUNGUSINFO, which contains the PATIENT_KEYs 1, 3, 8. Thus, we're left with PATIENT_KEYs 1 and 3 only.

Suppose we also want to keep all rows from PATFUNGUSINFO. Instead of an INNER JOIN, we do a RIGHT JOIN, like this:

```
SELECT A.PATIENT_KEY, A.PAT_GENDER, A.PAT_DEAD, A.PAT_WEIGHT,
       A.PAT_HEIGHT, B.PAT_ADDR, B.PAT_CITY, B.PAT_STATE, C.PAT_FUNGUS
 FROM PATIENTMASTER A
      INNER JOIN PATADDRINFO B
      ON A.PATIENT_KEY=B.PATIENT_KEY
       RIGHT JOIN PATFUNGUSINFO C
       ON A.PATIENT_KEY=C.PATIENT_KEY
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT	PAT_ADDR	PAT_CITY	PAT_STATE	PAT_FUNGUS
1	M	Y	130	70.52	123 Main St.	Philadelphia	PA	Y
3	F	N	230	91.56	567 Third St.	Blobby	DE	N
								Y

Take note of that last row. It contains the Y for PATIENT_KEY 8, but the PATIENT_KEY value does not display! Sup wit' dat? This is similar to several of our previous examples. Take note that we specifically asked for A.PATIENT_KEY and not C.PATIENT_KEY. We'll discuss this later on (later…always later!).

So we've used the ON Clause and you may be thinking that we never really need the WHERE Clause any more. This is not true, silly-billy! Think of the ON Clause as the way you tell the database **how to join tables together** whereas the WHERE Clause is the way you tell the database **how to subset the data** in the tables. For instance, let's pull only males from Pennsylvania with a foot fungus:

```
SELECT A.PATIENT_KEY, A.PAT_GENDER, A.PAT_DEAD, A.PAT_WEIGHT,
       A.PAT_HEIGHT, B.PAT_ADDR, B.PAT_CITY, B.PAT_STATE, C.PAT_FUNGUS
 FROM PATIENTMASTER A
      INNER JOIN PATADDRINFO B
      ON A.PATIENT_KEY=B.PATIENT_KEY
       RIGHT JOIN PATFUNGUSINFO C
       ON A.PATIENT_KEY=C.PATIENT_KEY
 WHERE A.PAT_GENDER='M'
       AND B.PAT_STATE='PA'
       AND C.PAT_FUNGUS='Y'
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT	PAT_ADDR	PAT_CITY	PAT_STATE	PAT_FUNGUS
1	M	Y	130	70.52	123 Main St.	Philadelphia	PA	Y

To wrap up this section, here's a SQL query involving everything we've learned so far:

```
SELECT A.PATIENT_KEY, A.PAT_GENDER, A.PAT_DEAD, A.PAT_WEIGHT,
       A.PAT_HEIGHT, B.PAT_ADDR, B.PAT_CITY, B.PAT_STATE, C.PAT_FUNGUS,
       703*PAT_WEIGHT/(PAT_HEIGHT*PAT_HEIGHT) AS BMI
 FROM PATIENTMASTER A
      INNER JOIN PATADDRINFO B
      ON A.PATIENT_KEY=B.PATIENT_KEY
       RIGHT JOIN PATFUNGUSINFO C
       ON A.PATIENT_KEY=C.PATIENT_KEY
 WHERE A.PAT_GENDER='M'
       AND B.PAT_STATE='PA'
       AND C.PAT_FUNGUS='Y'
       AND A.PATIENT_KEY IN (1,2,3,4,5,6)
       AND 703*PAT_WEIGHT/(PAT_HEIGHT*PAT_HEIGHT)>=10
 ORDER BY A.PAT_GENDER,A.PAT_WEIGHT DESC
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT	PAT_ADDR	PAT_CITY	PAT_STATE	PAT_FUNGUS	BMI
1	M	Y	130	70.52	123 Main St.	Philadelphia	PA	Y	18.37

Our Third SQL DML Example (GROUP BY/HAVING)

Pulling a subset of data from one or more database tables, joining them together and sorting them is all well and good, but one of the most important uses of SQL is to answer business questions. This usually, although not always, involves things like counting the number of patients with foot fungus, computing total weight of the male versus female patients, determining the height of the smallest male and female, determining the average weight of the living versus dead patients, etc. Naturally, these morbid business questions reflect our three tables, but real-world examples are similar.

So, let's take a look at a SQL query that counts the number of patients with a foot fungus:

```
SELECT COUNT(A.PATIENT_KEY) as FUNGUS_PATIENT_COUNT
 FROM PATIENTMASTER A
      INNER JOIN PATFUNGUSINFO B
      ON A.PATIENT_KEY=B.PATIENT_KEY
 WHERE B.PAT_FUNGUS='Y'

FUNGUS_PATIENT_COUNT
                   1
```

Here we've used the COUNT() **aggregate function** to count the number of PATIENT_KEYs returned in the SQL query. Besides the COUNT() aggregate function, we have the following additional aggregate functions:

- ☐ COUNT(*column*) – Counts the number of items in *column*
- ☐ SUM(*column*) – Sums up the values in the *column*
- ☐ AVG(*column*) – Computes the average of the values in the *column*
- ☐ MIN(*column*) – Determines the minimum value in *column*
- ☐ MAX(*column*) – Determines the maximum value in *column*

Note that some databases may offer additional aggregate functions, but the ones listed above are available *on da street*.

In the SQL code above, the COUNT() function will count the number of non-NULL values across the column PATIENT_KEY, but suppose we want to know the number of distinct non-NULL values instead. To do that, you can combine the DISTINCT keyword with the COUNT() function:

```
SELECT COUNT(DISTINCT A.PATIENT_KEY) as FUNGUS_PATIENT_COUNT
 FROM PATIENTMASTER A
      INNER JOIN PATFUNGUSINFO B
      ON A.PATIENT_KEY=B.PATIENT_KEY
 WHERE B.PAT_FUNGUS='Y'

FUNGUS_PATIENT_COUNT
                   1
```

Here we used the COUNT() aggregate function along with the DISTINCT keyword to count the number of unique PATIENT_KEYs in the PATIENTMASTER table. Note that you should use the DISTINCT keyword within the COUNT() aggregate function: COUNT(**DISTINCT** A.PATIENT_KEY). If not, you'll be counting non-NULL occurrences of A.PATIENT_KEY instead of unique counts.

Instead of placing a column name within the COUNT() aggregate function, you can use an asterisk (*) instead. This will count the number of rows in the table instead:

```
SELECT COUNT(*) as ROWS_IN_PATIENTMASTER
 FROM PATIENTMASTER A

ROWS_IN_PATIENTMASTER
                    6
```

Note that if a PATIENT_KEY is NULL, it won't be counted in the second SQL example above, but **will be** counted in the third example. This is true for the other aggregate functions as well (AVG(), etc.).

Now, let's sum up the total weight in pounds across the patients in the PATIENTMASTER table as well as determine the minimum and maximum weight:

```
SELECT SUM(A.PAT_WEIGHT) as TOTAL_FATNESS,
       MIN(A.PAT_WEIGHT) as MIN_FATNESS,
       MAX(A.PAT_WEIGHT) as MAX_FATNESS
 FROM PATIENTMASTER A

TOTAL_FATNESS MIN_FATNESS MAX_FATNESS
         1530         130         380
```

So far, we've computed information across all of the rows in the PATIENTMASTER table. But, what happens if we want to determine the total, minimum and maximum weight by PAT_GENDER? To do this, we have to use the GROUP BY Clause, like this:

```
SELECT A.PAT_GENDER,
       SUM(A.PAT_WEIGHT) as TOTAL_FATNESS,
       MIN(A.PAT_WEIGHT) as MIN_FATNESS,
       MAX(A.PAT_WEIGHT) as MAX_FATNESS
 FROM PATIENTMASTER A
 GROUP BY A.PAT_GENDER

PAT_GENDER TOTAL_FATNESS MIN_FATNESS MAX_FATNESS
M                    790         130         380
F                    740         180         330
```

Take note that we used the same column on the SELECT Clause as we did on the GROUP BY Statement. Be neat and tidy with your SQL code: place the columns used on the GROUP BY Clause in the same order on the SELECT Clause!! People will love you for it!

Next, let's determine the minimum and maximum weight of the males and females, living or dead:

```
SELECT A.PAT_GENDER,
       A.PAT_DEAD,
       SUM(A.PAT_WEIGHT) as TOTAL_FATNESS,
       MIN(A.PAT_WEIGHT) as MIN_FATNESS,
       MAX(A.PAT_WEIGHT) as MAX_FATNESS
 FROM PATIENTMASTER A
 GROUP BY A.PAT_GENDER,A.PAT_DEAD

PAT_GENDER PAT_DEAD TOTAL_FATNESS MIN_FATNESS MAX_FATNESS
M          Y                  130         130         130
M          N                  660         280         380
F          N                  410         180         230
F          Y                  330         330         330
```

As you can see, the GROUP BY Clause summarizes your data down to the distinct levels of the values in the GROUP BY columns: PAT_GENDER and PAT_DEAD. For those more familiar with Microsoft Excel, this should have a very *pivot table* vibe to it.

Recall that we used a WHERE Clause to pull a subset of data. Taking a look at the results above, suppose we want to keep only those rows having TOTAL_FATNESS greater than or equal to 400. In the output above, row number 2 and 3 fit the bill here. But, we can't use the WHERE Clause in this case because the WHERE Clause is performed prior to the GROUP BY Clause being performed. What we really need (in a *hope-y* and *wish-y* fashion) is a WHERE Clause that can be applied to the results of the SQL query **after** the GROUP BY Clause has completed. The

HAVING Clause is just what we need to get the job done! You can think of a HAVING Clause as a WHERE Clause applied **after** the GROUP BY Clause has finished summarizing the data. For example,

```
SELECT A.PAT_GENDER,
        A.PAT_DEAD,
        SUM(A.PAT_WEIGHT) as TOTAL_FATNESS,
        MIN(A.PAT_WEIGHT) as MIN_FATNESS,
        MAX(A.PAT_WEIGHT) as MAX_FATNESS
 FROM PATIENTMASTER A
 GROUP BY A.PAT_GENDER,A.PAT_DEAD
 HAVING SUM(A.PAT_WEIGHT) >= 400
```

PAT_GENDER	PAT_DEAD	TOTAL_FATNESS	MIN_FATNESS	MAX_FATNESS
M	N	660	280	380
F	N	410	180	230

Take note that we didn't use the column alias TOTAL_FATNESS, but instead we used SUM(A.PAT_WEIGHT). This is because the three column aliases TOTAL_FATNESS, MIN_FATNESS and MAX_FATNESS are assigned **after the SQL query has completed** and, thus, can't be used while the query is *in-flight*.

To wrap this section up, here's an example using everything we've learned so far:

```
SELECT A.PAT_GENDER,C.PAT_FUNGUS,
       AVG(703*PAT_WEIGHT/(PAT_HEIGHT*PAT_HEIGHT)) AS AVG_BMI,
       COUNT(DISTINCT A.PATIENT_KEY) AS DISTINCT_PATS,
       COUNT(*) AS NBR_OF_ROWS
 FROM PATIENTMASTER A
      INNER JOIN PATADDRINFO B
      ON A.PATIENT_KEY=B.PATIENT_KEY
       RIGHT JOIN PATFUNGUSINFO C
       ON A.PATIENT_KEY=C.PATIENT_KEY
 WHERE A.PAT_GENDER='M'
       AND B.PAT_STATE='PA'
       AND C.PAT_FUNGUS='Y'
       AND A.PATIENT_KEY IN (1,2,3,4,5,6)
 GROUP BY A.PAT_GENDER,C.PAT_FUNGUS
 HAVING AVG(703*PAT_WEIGHT/(PAT_HEIGHT*PAT_HEIGHT))>1.0
 ORDER BY A.PAT_GENDER,C.PAT_FUNGUS DESC
```

PAT_GENDER	PAT_FUNGUS	AVG_BMI	DISTINCT_PATS	NBR_OF_ROWS
M	Y	18.3769769	1	1

Our Fourth SQL DML Example (Subqueries)

Recall that we showed an example above using the IN() Statement:

```
SELECT *
 FROM PATIENTMASTER
 WHERE PATIENT_KEY IN (1,3,5)
```

But, suppose you already had a table containing a list of desired PATIENT_KEYs. In this case, you should use an INNER JOIN, but you can also use a *subquery*. Let's take a look at the INNER JOIN syntax first. Assume that we have a table called DESIREDPATIENTS containing a single column PATIENT_KEY with three patient key values 1, 3 and 5 as its rows. Let's use an INNER JOIN against the PATIENTMASTER table:

```
SELECT A.*
 FROM PATIENTMASTER A INNER JOIN DESIREDPATIENTS B
 ON A.PATIENT_KEY=B.PATIENT_KEY
 ORDER BY A.PATIENT_KEY

PATIENT_KEY PAT_GENDER PAT_DEAD PAT_WEIGHT PAT_HEIGHT
          1 M          Y               130      70.52
          3 F          N               230      91.56
          5 F          Y               330      112.6
```

Instead of using an `INNER JOIN`, you can use a subquery along with the `IN()` Statement, like this:

```
SELECT A.*
 FROM PATIENTMASTER A
 WHERE A.PATIENT_KEY IN (SELECT DISTINCT PATIENT_KEY
                          FROM DESIREDPATIENTS)
```

Note that subqueries can be useful when it's difficult to come up with a SQL query not involving subqueries. (A truer statement has never been said!) Think of a subquery as tiny SQL query. As we've seen, we can use a subquery in a `WHERE` Clause with the `IN()` Statement, but you can also use subqueries in place of tables themselves:

```
SELECT A.PATIENT_KEY,A.PAT_GENDER,B.PAT_FUNGUS
 FROM (SELECT PATIENT_KEY,PAT_GENDER
        FROM PATIENTMASTER) A INNER JOIN (SELECT PATIENT_KEY,PAT_FUNGUS
                                           FROM PATFUNGUSINFO
                                           WHERE PAT_FUNGUS='Y') B
 ON A.PATIENT_KEY=B.PATIENT_KEY
```

Subqueries can be as simple or as complicated as you want (or need). If you find that your subqueries are becoming too complicated, or just visually unappealing, you can remove them and place them in a `WITH` Clause just above the `SELECT` Clause. Let's rework the example above, but this time using a `WITH` Clause for neatness:

```
WITH vwPATGENDER AS (SELECT PATIENT_KEY,PAT_GENDER
                      FROM PATIENTMASTER),
     vwPATFUNGUS AS (SELECT PATIENT_KEY,PAT_FUNGUS
                      FROM PATFUNGUSINFO
                      WHERE PAT_FUNGUS='Y')
SELECT A.PATIENT_KEY,A.PAT_GENDER,B.PAT_FUNGUS
 FROM vwPATGENDER A INNER JOIN vwPATFUNGUS B
 ON A.PATIENT_KEY=B.PATIENT_KEY
```

The `WITH` Clause allows you to move your subqueries off the `FROM` Clause and into their own clause. Take note that each subquery is given its own name: `vwPATGENDER` and `vwPATFUNGUS`. These names are then used in place of table names on the `FROM` Clause. Ahhh! Now that's a-nice a-neat SQL code!!

Let's end this section with an example showing off the stuff we've learned so far:

```
WITH vwPATMSTR AS (SELECT *
                    FROM PATIENTMASTER
                    WHERE PAT_GENDER='M'
                          AND PATIENT_KEY IN (SELECT DISTINCT PATIENT_KEY
                                               FROM PATIENTMASTER)),
     vwPATADDR AS (SELECT *
                    FROM PATADDRINFO
                    WHERE PAT_STATE='PA'),
     vwPATFUN  AS (SELECT *
                    FROM PATFUNGUSINFO
                    WHERE PAT_FUNGUS='Y')
```

```
SELECT A.PAT_GENDER,C.PAT_FUNGUS,
       AVG(703*PAT_WEIGHT/(PAT_HEIGHT*PAT_HEIGHT)) AS AVG_BMI,
       COUNT(A.PATIENT_KEY) AS DISTINCT_PATS
 FROM vwPATMSTR A
      INNER JOIN vwPATADDR B
      ON A.PATIENT_KEY=B.PATIENT_KEY
       RIGHT JOIN vwPATFUN C
       ON A.PATIENT_KEY=C.PATIENT_KEY
 GROUP BY A.PAT_GENDER,C.PAT_FUNGUS
 HAVING AVG(703*PAT_WEIGHT/(PAT_HEIGHT*PAT_HEIGHT))>1.0
 ORDER BY A.PAT_GENDER,C.PAT_FUNGUS DESC
```

PAT_GENDER	PAT_FUNGUS	AVG_BMI	DISTINCT_PATS
M	Y	18.3769769	1

Our Fifth SQL DML Example (CASE Statement and Functions)

There may be times when you want to create a new column based on the values of one or more existing columns. For example, let's rank the patients in the PATIENTMASTER table such that:

- ☐ Male patients who weigh less than or equal to 250 pounds are coded as a 1
- ☐ Male patients who weigh more than 250 pounds are coded as a 2
- ☐ Female patients who weigh less than or equal to 250 pounds are coded as a 3
- ☐ Female patients who weigh more than 250 pounds are coded as a 4

To do this, we introduce the CASE Statement. Here's our SQL query:

```
SELECT PATIENT_KEY,
       PAT_GENDER,
       PAT_WEIGHT,
       CASE
        WHEN PAT_GENDER='M' AND PAT_WEIGHT <= 250 THEN 1
        WHEN PAT_GENDER='M' AND PAT_WEIGHT >  250 THEN 2
        WHEN PAT_GENDER='F' AND PAT_WEIGHT <= 250 THEN 3
        WHEN PAT_GENDER='F' AND PAT_WEIGHT >  250 THEN 4
        ELSE                                           5
       END AS PAT_WEIGHT_CODING
 FROM PATIENTMASTER
 ORDER BY 4
```

PATIENT_KEY	PAT_GENDER	PAT_WEIGHT	PAT_WEIGHT_CODING
1	M	130	**1**
4	M	280	**2**
6	M	380	**2**
2	F	180	**3**
3	F	230	**3**
5	F	330	**4**

You can also place the CASE Statement **within** an aggregate function. This allows for more possibilities, and dare I say, more fun! Here's an example:

```
SELECT PAT_GENDER,
       SUM(
           CASE
            WHEN PAT_WEIGHT < 250 THEN 1
            ELSE                       0
           END
          ) AS NBR_PATS_UNDER_250
 FROM PATIENTMASTER
 GROUP BY PAT_GENDER
```

In this case, we're summing up the 1s and 0s produced by the CASE Statement, not the PAT_WEIGHT. Note that you don't need a column alias within the function.

Many databases offer additional functions you can use in your SQL queries. These functions can do things like substring a text column, raise a number to a power, take the absolute value, compute the ceiling or floor, determine the length of a text column, trim off the blanks from a text column, round a number to a specific decimal point, and so on.

For example, let's determine the length of the patient's address field from the PATADDRINFO table by using the LENGTH() function:

```
SELECT PAT_ADDR,LENGTH(PAT_ADDR) AS ADDR_LENGTH
 FROM PATADDRINFO
```

PAT_ADDR	ADDR_LENGTH
123 Main St.	12
234 Second St.	14
567 Third St.	13
890 Fourth St.	14

Let's retrieve the house number, luckily always the first three digits, from PAT_ADDR by using the SUBSTR() function:

```
SELECT PAT_ADDR,SUBSTR(PAT_ADDR,1,3) AS HOUSE_NBR
 FROM PATADDRINFO
```

PAT_ADDR	HOUSE_NBR
123 Main St.	123
234 Second St.	234
567 Third St.	567
890 Fourth St.	890

Recall that in order to compute the body mass index (BMI), we had to multiply the PAT_HEIGHT by itself to simulate raising to the second power. Screw that! Let's use the POWER() function to raise PAT_HEIGHT to the second power like big boys and girls:

```
SELECT PATIENT_KEY,
       PAT_GENDER,
       PAT_DEAD,
       PAT_WEIGHT,
       PAT_HEIGHT,
       703*PAT_WEIGHT/(PAT_HEIGHT*PAT_HEIGHT) AS BMI_OLD_WAY,
       703*PAT_WEIGHT/POWER(PAT_HEIGHT,2) AS BMI_NEW_WAY
 FROM PATIENTMASTER
```

PATIENT_KEY	PAT_GENDER	PAT_DEAD	PAT_WEIGHT	PAT_HEIGHT	BMI_OLD_WAY	BMI_NEW_WAY
1	M	Y	130	70.52	18.3769769	18.3769769
2	F	N	180	81.04	19.2676596	19.2676596
5	F	Y	330	112.6	18.2975307	18.2975307

```
          6 M          N                380     123.12  17.6230758  17.6230758
          3 F          N                230      91.56   19.287307   19.287307
          4 M          N                280     102.08  18.8900033  18.8900033
```

Boy, there are a lot of decimal places in those BMI numbers! Let's round to the second decimal place using the `ROUND()` function:

```
SELECT PATIENT_KEY,
       PAT_GENDER,
       PAT_DEAD,
       PAT_WEIGHT,
       PAT_HEIGHT,
       ROUND(703*PAT_WEIGHT/POWER(PAT_HEIGHT,2),2) AS BMI_NEW_WAY_ROUNDED
 FROM PATIENTMASTER

PATIENT_KEY PAT_GENDER PAT_DEAD PAT_WEIGHT PAT_HEIGHT BMI_NEW_WAY_ROUNDED
          1 M          Y               130      70.52               18.38
          2 F          N               180      81.04               19.27
...snip...
```

Let's end this section with a comprehensive example:

```
WITH vwPATMSTR AS (SELECT *
                   FROM PATIENTMASTER
                   WHERE PAT_GENDER IN ('M','F')
                         AND PATIENT_KEY IN (SELECT DISTINCT PATIENT_KEY
                                             FROM PATIENTMASTER)),
     vwPATADDR AS (SELECT *
                   FROM PATADDRINFO),
     vwPATFUN  AS (SELECT *
                   FROM PATFUNGUSINFO
                   WHERE PAT_FUNGUS IN ('N','Y'))
SELECT A.PAT_GENDER,C.PAT_FUNGUS,
       CASE
        WHEN ROUND(AVG(703*PAT_WEIGHT/POWER(PAT_HEIGHT,2)),2) <= 19
                                                              THEN 'TEENSY'
        WHEN ROUND(AVG(703*PAT_WEIGHT/POWER(PAT_HEIGHT,2)),2) >  19
                                                              THEN 'FAATSY'
       END AS AVG_BMI_RANK
 FROM vwPATMSTR A
      INNER JOIN vwPATADDR B
      ON A.PATIENT_KEY=B.PATIENT_KEY
       RIGHT JOIN vwPATFUN C
       ON A.PATIENT_KEY=C.PATIENT_KEY
 GROUP BY A.PAT_GENDER,C.PAT_FUNGUS
 HAVING AVG(703*PAT_WEIGHT/POWER(PAT_HEIGHT,2))>1.0

PAT_GENDER PAT_FUNGUS AVG_BM
F          N          FAATSY
M          Y          TEENSY
```

Our Sixth SQL DML Example (UNION/INTERSECT/MINUS/EXCEPT)

If you recall your elementary school set theory (which you learned just after having milk and cookies), you'll remember that the concepts of set union, set intersection and set minus were presented. You won't be surprised that union, intersection and minus are available in SQL as well since tables can be considered sets. Just as a reminder, here's the definition of union, intersection and minus.

Given the sets `A={bananas,oranges,kiwi,grapes}` and `B={grapes,kaluha,vodka,gin}`:

<u>Union</u>

```
A∪B = {bananas,oranges,kiwi,grapes,kaluha,vodka,gin}
```

<u>Intersection</u>

```
A∩B = {grapes}
```

<u>Minus/Except</u>

```
A\B = {bananas,oranges,kiwi}
```

To code a union in SQL, do this:

```
SELECT FRUIT FROM SETA
UNION
SELECT FRUIT FROM SETB
```

To code an intersection in SQL, do this:

```
SELECT FRUIT FROM SETA
INTERSECT
SELECT FRUIT FROM SETB
```

To code a minus (or except) in SQL, do this:

```
SELECT FRUIT FROM SETA
MINUS
SELECT FRUIT FROM SETB
```

Note: Some SQL flavors, like HiveQL and ImpalaSQL, do not have the `INTERSECT` and `MINUS/EXCEPT` keywords.

Note that `UNION` removes duplicate values automatically (same as in set theory). To keep the duplicates, place the keyword `ALL` after `UNION`:

```
SELECT FRUIT FROM SETA
UNION ALL
SELECT FRUIT FROM SETB
```

An example of one place you could use `UNION ALL` is when you want to bring together two (or more) separate sets of data. For example, we could compute the BMI on the males only and then females only from the `PATIENTMASTER` table and the use `UNION ALL` to bring everything together:

```
SELECT PATIENT_KEY,703*PAT_WEIGHT/POWER(PAT_HEIGHT,2) AS BMI
 FROM PATIENTMASTER
 WHERE PAT_GENDER='M'
UNION ALL
SELECT PATIENT_KEY,703*PAT_WEIGHT/POWER(PAT_HEIGHT,2) AS BMI
 FROM PATIENTMASTER
 WHERE PAT_GENDER='F'
```

Let's end this section with a comprehensive example.

```
WITH vwPATMSTR AS (SELECT *
                    FROM PATIENTMASTER
                    WHERE PAT_GENDER IN ('M','F')
                          AND PATIENT_KEY IN (SELECT DISTINCT PATIENT_KEY
                                               FROM PATIENTMASTER)),
     vwPATADDR AS (SELECT *
                    FROM PATADDRINFO),
     vwPATFUN  AS (SELECT *
                    FROM PATFUNGUSINFO
                    WHERE PAT_FUNGUS IN ('N','Y'))
SELECT 'ALL GENDERS' AS TITLE,A.PAT_GENDER,C.PAT_FUNGUS,
       CASE
        WHEN ROUND(AVG(703*PAT_WEIGHT/POWER(PAT_HEIGHT,2)),2) <= 19
                                                              THEN 'TEENSY'
        WHEN ROUND(AVG(703*PAT_WEIGHT/POWER(PAT_HEIGHT,2)),2) >  19
                                                              THEN 'FAATSY'
       END AS AVG_BMI_RANK
 FROM vwPATMSTR A
      INNER JOIN vwPATADDR B
      ON A.PATIENT_KEY=B.PATIENT_KEY
       RIGHT JOIN vwPATFUN C
       ON A.PATIENT_KEY=C.PATIENT_KEY
 GROUP BY A.PAT_GENDER,C.PAT_FUNGUS
 HAVING AVG(703*PAT_WEIGHT/POWER(PAT_HEIGHT,2))>1.0
UNION ALL
SELECT 'MALE GENDER' AS TITLE,A.PAT_GENDER,C.PAT_FUNGUS,
       CASE
        WHEN ROUND(AVG(703*PAT_WEIGHT/POWER(PAT_HEIGHT,2)),2) <= 19
                                                              THEN 'TEENSY'
        WHEN ROUND(AVG(703*PAT_WEIGHT/POWER(PAT_HEIGHT,2)),2) >  19
                                                              THEN 'FAATSY'
       END AS AVG_BMI_RANK
 FROM vwPATMSTR A
      INNER JOIN vwPATADDR B
      ON A.PATIENT_KEY=B.PATIENT_KEY
       RIGHT JOIN vwPATFUN C
       ON A.PATIENT_KEY=C.PATIENT_KEY
 WHERE A.PAT_GENDER='M'
 GROUP BY A.PAT_GENDER,C.PAT_FUNGUS
 HAVING AVG(703*PAT_WEIGHT/POWER(PAT_HEIGHT,2))>1.0
```

Our Seventh SQL DML Example (Correlated Subqueries and EXISTS)

Recall that a subquery is just a SQL query surrounded by parentheses and used in place of, say, an `IN()` list or a table. For example,

```
SELECT A.*
 FROM PATIENTMASTER A
 WHERE A.PATIENT_KEY IN (SELECT DISTINCT PATIENT_KEY
                          FROM DESIREDPATIENTS)
```

In this case, the emboldened text is the subquery. Take note that none of the columns from `A` appear within the subquery. If one or more columns do appear within the subquery, the subquery is referred to as a *correlated subquery*. For example, here we're returning a distinct list of patients who don't have a fungus:

```
SELECT DISTINCT A.PATIENT_KEY
 FROM PATIENTMASTER A
 WHERE A.PATIENT_KEY IN (SELECT B.PATIENT_KEY
                          FROM PATFUNGUSINFO B
                          WHERE B.PATIENT_KEY=A.PATIENT_KEY
                                AND B.PAT_FUNGUS='N')

PATIENT_KEY
          1
```

As you see in the SQL code above, `A.PATIENT_KEY` is being referenced within the subquery. The way you can think about this is at a *row-by-row* level. That is, for each row in the `PATIENTMASTER`, the correlated subquery is executed by replacing `A.PATIENT_KEY` with the actual value of the `PATIENT_KEY` in the current row in the `PATIENTMASTER`. The `WHERE` Clause is then evaluated and a row from the `PATIENTMASTER` is returned if the `WHERE` Clause is true.

In the SQL code above, we used a correlated subquery to determine whether to return rows of data from the `PATIENTMASTER` table where the patient did not have a foot fungus. In some cases, you may not care about what data is returned from the subquery, just that there is data at all in the subquery. In these instances, you can use the `EXISTS` condition. For example,

```
SELECT DISTINCT A.PATIENT_KEY
 FROM PATIENTMASTER A
 WHERE EXISTS (SELECT B.PATIENT_KEY
               FROM PATFUNGUSINFO B
               WHERE B.PATIENT_KEY=A.PATIENT_KEY
                     AND B.PAT_FUNGUS='Y')
PATIENT_KEY
          1
```

Note that some people replace `B.PATIENT_KEY` on the subquery `SELECT` Clause with either `*`, `1` or `NULL`. Since you're just testing for existence, there's no need to return a value (although `NULL` is not very intuitive).

Now, you can get a list of patients who do not have a foot fungus using the `NOT EXISTS` concept:

```
SELECT DISTINCT A.PATIENT_KEY
 FROM PATIENTMASTER A
 WHERE NOT EXISTS (SELECT 1
                   FROM PATFUNGUSINFO B
                   WHERE B.PATIENT_KEY=A.PATIENT_KEY
                         AND B.PAT_FUNGUS='Y')
PATIENT_KEY
          6
          2
          5
          4
          3
```

Our Eighth SQL DML Example (LIKE)

Occasionally, you'll want to subset your data based on data within a text string. For example, you may want all drugs in tablet form for which there's an amount in milligrams specified by `MG`:

```
SELECT BRAND,LABEL
 FROM DRUG_MASTER
 WHERE LABEL LIKE '%MG%TABLET%'
```

```
BRAND   LABEL
SIMPLEX SIMPLEX 5MG TABLET
SIMPLEX SIMPLEX 10MG TABLET
SIMPLEX SIMPLEX 15MG TABLET SAMPLE
```

As you see above, the percent sign (`%`) is being used as a wildcard. This has the effect of matching zero or more characters. Note that this does not match a `NULL` value!

You can also use the underscore (_) to match exactly **one** character. Note that this does not match a `NULL` value!

```
SELECT BRAND,LABEL
 FROM DRUG_MASTER
 WHERE LABEL LIKE '% _MG TABLET%';
```

```
BRAND   LABEL
SIMPLEX SIMPLEX 5MG TABLET
```

SQL Data Definition Language (DDL)

In this section, let's concentrate on the SQL Data Definition Language, or DDL.

What is SQL DDL?

SQL Data Definition Language, DDL, is used to create/drop a table, truncate the data within a table, delete specific rows within a table, etc. DDL is, in general, used to **manage**/**modify** database tables, whereas DML is used to **query** database tables.

Creating a database table involves assigning a name to the table as well as specifying the columns in the table. You use the `CREATE TABLE` Statement to create a table. For example, here's the specification for `PATIENTMASTER`:

```
Name           Type
 PATIENT_KEY    BIGINT
 PAT_GENDER     STRING
 PAT_DEAD       STRING
 PAT_WEIGHT     DOUBLE
 PAT_HEIGHT     DOUBLE
```

The first column lists the names of the columns in the `PATIENTMASTER` table. The second column tells you the data type associated with each column. The data type lets the database know whether the column will be filled with numeric values, character values, date and time values, etc.

When you no longer want the data within a table, but you still want the table to hang around, you can truncate the table; that is, remove all of the data from the table. For this, use the `TRUNCATE TABLE` Statement.

If you want to remove specific rows from a table then you delete from the table. The `DELETE` command is used to delete specific rows from a table.

Note that you can use the `DELETE` Statement to remove all data from a table, but this tends to be slower than the `TRUNCATE` Statement.

You can insert additional rows into a database table by using the `INSERT INTO` Statement.

In many databases, after you create a table and insert rows into that table, it's a good idea to compute some useful statistics on that table in order for the database to better know how to query the table. The better the statistics, the faster the query will run. The worse or non-existent the statistics, the query may run longer than with good

statistics. In ImpalaSQL, you can use the `COMPUTE STATS` command to gather statistics. We talk more about this later in the book.

Our First SQL DDL Example (CREATE/DROP/TRUNCATE)

In this section, we learn how to create and drop tables. Recall we mentioned that in order to create a table, you need to have a table name in mind as well as know what the columns will be.

A table name can be any valid name. Some databases limit your table names to a specific maximum number of characters. For example, Oracle limits you to 30 characters, SAS to 32 characters and Microsoft SQL Server to 128 characters. Personally, I stick to 30 characters or less for table names.

Once you know your table name, you need to figure out the columns that will make up that table and give them names as well. Just like for table names, it's best to stick to column names 30 characters or less.

Finally, once you know the column names, you need to determine the data type for each column. A data type lets the database know whether the column is to be filled with numeric data (like `1234` or `3.1415`), with character data (like `Spineless twit` or `foul odor`), or dates (like `1963-03-01` or `01MAR1963`). We discuss the ImpalaSQL data types later in the book.

For example, let's create the `PATIENTMASTER` table:

```
CREATE TABLE PATIENTMASTER(PATIENT_KEY BIGINT,
                           PAT_GENDER  STRING,
                           PAT_DEAD    STRING,
                           PAT_WEIGHT  DOUBLE,
                           PAT_HEIGHT  DOUBLE)
```

That's all there is to it! Notice that you specify the table name right after `CREATE TABLE`, followed by a comma-delimited list of columns with their associated data types. For the column `PATIENT_KEY`, we assigned the data type `BIGINT`; the two text columns, we assigned `STRING`; and the two numeric columns, we assigned to `DOUBLE`. Let's say that we wanted to have a birth date column, `DATE_OF_BIRTH`, in this table. Here's the new SQL DDL:

```
CREATE TABLE PATIENTMASTER_WITH_DOB(PATIENT_KEY   BIGINT,
                                    PAT_GENDER    STRING,
                                    PAT_DEAD      STRING,
                                    PAT_WEIGHT    DOUBLE,
                                    PAT_HEIGHT    DOUBLE,
                                    DATE_OF_BIRTH TIMESTAMP)
```

After you've created the table, if you want to remove the table definition as well as the associated data, use the `DROP TABLE` command:

```
DROP TABLE PATIENTMASTER_WITH_DOB
```

Again, be aware that the data will be deleted as well as the table definition! Boom! Gone!!

If you want to remove all of the data from a table without dropping the table, then use the `TRUNCATE` command:

```
TRUNCATE TABLE PATIENTMASTER_WITH_DOB
```

Our Second SQL DDL Example (DELETE/INSERT)

As mentioned above, the `TRUNCATE` command removes **all** of the data from a table. But, there may be times when you want to remove only specific rows. To do this, use the `DELETE` command instead of the `TRUNCATE` command. For example, to delete all of the Males from the `PATIENTMASTER` table, use this syntax:

```
DELETE
 FROM PATIENTMASTER
 WHERE PAT_GENDER='M'
```

Once you've used the `CREATE TABLE` command to create the table, you can use the `INSERT` command to load the table with data. The `INSERT` command comes in two flavors, one allows you to specify a SQL Query, and the other allows you to enter hard-coded values. Note that some DBMSs require the `COMMIT` command to be submitted after the `INSERT` command to ensure the data is actually loaded into the table. This is not true for ImpalaSQL.

For example, let's create a table called `GENDER` which contains all of the gender descriptions:

```
CREATE TABLE GENDER(PAT_GENDER VARCHAR2(1),
                    GENDER_DESC VARCHAR2(50))
```

Next, let's enter in the hard-coded values:

```
INSERT INTO GENDER VALUES('M','Male')
INSERT INTO GENDER VALUES('F','Female')
INSERT INTO GENDER VALUES('U','Unknown')
```

Note that after the `VALUES` keyword, you enter a comma-delimited list of values in the same column order as table definition. To insert data into your table based on a SQL Query, use the following syntax:

```
CREATE TABLE PATIENTMASTER_BACKUP(PATIENT_KEY BIGINT,
                                  PAT_GENDER  STRING,
                                  PAT_DEAD    STRING,
                                  PAT_WEIGHT  DOUBLE,
                                  PAT_HEIGHT  DOUBLE)

INSERT INTO PATIENTMASTER_BACKUP
 SELECT PATIENT_KEY,PAT_GENDER,PAT_DEAD,PAT_WEIGHT,PAT_HEIGHT
  FROM PATIENTMASTER
```

As you can see from the example above, we're making a backup table from the original. An easier way to make a new table based on a SQL query from a pre-existing table is to use the `CREATE TABLE AS` (`CTAS`) syntax, like this:

```
CREATE TABLE PATIENTMASTER_BACKUP AS
 SELECT *
  FROM PATIENTMASTER
```

Our Third SQL DDL Example (UPDATE)

Note: When using ImpalaSQL, for the most part, only the KUDU storage format can be updated. The KUDU storage format is explained in more detail later on in the book.

Rather than reloading all or some of the data into a table if you've found a mistake, you can use the `UPDATE` statement to modify the values of one or more columns for one or more rows. Note that some DBMSs require the `COMMIT` command to be submitted after the `UPDATE` command to ensure the table is actually updated. This is not true for ImpalaSQL.

For example, let's assume you found out that the patient assigned to `PATIENT_KEY=6` is really dead (`PAT_DEAD='Y'`). Naturally, we want to update the `PAT_DEAD` column for that stiff. Here's how we do that using the `UPDATE` statement:

```
UPDATE PATIENTMASTER
 SET PAT_DEAD='Y'
 WHERE PATIENT_KEY=6
```

Notice that the `WHERE` Clause will limit the update to only those rows where `PATIENT_KEY` is `6`. If you don't specify the `WHERE` Clause, all of the rows will be updated…not good!!

Chapter 7 – Querying the Hadoop Database (Hue and SQL Clients)

Before we get hot and heavy into ImpalaSQL, let's take a quick look at the Hadoop User Experience (HUE) web interface as well as one of the SQL clients. Recall, in *Chapter 3 – Recommended Windows Client Software*, we installed and set up several SQL Clients, so you should be good to go for this chapter. (If you and your team chose a different SQL client, please let me know and I'll add it to the next edition of the book. Smoochies and huggies in advance, poppets!)

Hue – The Hadoop User Experience Web Interface

For some users, a full-blown SQL client isn't necessary, but being able to query the database occasionally is a good thing. For those people, Hue allows access to the Hadoop database from a web browser. The URL for Hue should be one of the responses to the Hadoop Administrator E-Mail.

Below is an e-mail you can send to those members of your team (or colleagues outside of your fab department) with instructions on how to use Hue from a web browser. Please change the e-mail to reflect your Hue URL, production schema (recall that I'm calling it `prod_schema`), linguistic sensibility, silly corporate policy, etc. etc.

Colleagues:

As many of you are aware by now, the *<insert dept name here>* department has moved off of the legacy *<insert legacy database name here>* database to our new technologically-advanced Hadoop database. With great advancements come great features such as faster query runtimes, the ability to run more detailed analytics and much, much more.

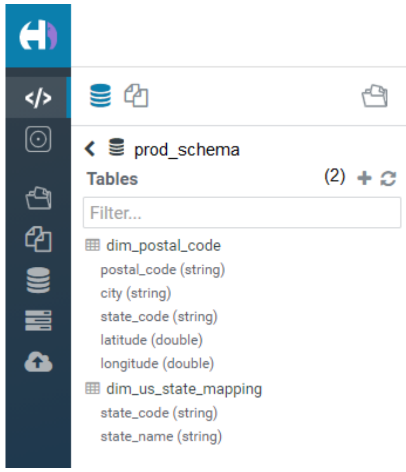
But, the feature that will impress you the most is the database web interface Hue. This gives you the ability to run SQL queries against the Hadoop database yourself using nothing but a web browser...all without having to wait in line for my department to run your requests. Naturally, more complicated requests should proceed through our standard request process. Below, we introduce Hue, but in the next few weeks, we'll have a meeting to discuss how to use Hue as well as work with the tables in our schema (named prod_schema) in more detail.

To access Hue, follow these instructions:

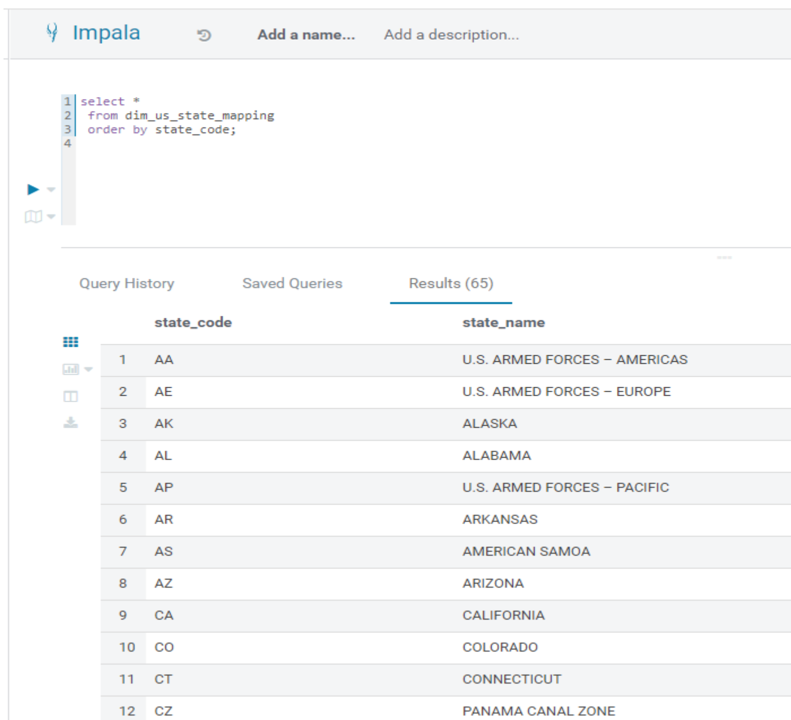
1. Start your web browser.
2. Insert the following URL in the Address Bar at the top of the web browser:

   http://*<insert hue url here>*/hue/accounts/login

3. When the Sign In web page appears, enter your Windows username and password into the appropriate input boxes and click the Sign In button. Note that your username/password should be the same as those used to log into your company laptop each morning.
4. On the left side, click Impala under Sources.
5. On the left side, click the schema name prod_schema under Databases.
6. At this point, on the left side, you'll see a list of tables you can query. If you click on a table, the columns will appear just below the table name. You should see something like this:



7. To query a table, enter a SQL query in the textbox at the top-center of the web page. Then, click the arrow to the left of the textbox to run the query. The results will appear below the SQL query textbox. You will see something like this:



8. You can export these results by clicking on the Export results button (last button to the left of the query results) and selecting the desired option from the popup menu: CSV, Excel, Clipboard, Export. Your results will be available almost immediately.

Any additional questions, please feel free to contact me at any time.

Thanks,
Bob

There are a few additional features of Hue I'd like to mention and you can modify the e-mail above to include them, if you feel it necessary; otherwise, these features can wait until your training session (sorry about adding more work to your already busy schedule!):

1. You can name and save a query in Hue by clicking on the floppy disk icon at the top-right. Remember, though, that Hue is not a professional SQL client and shouldn't be used to run production queries. That's what a SQL client and the Linux edge node server are for.
2. Although you can access tables in the Hadoop database without specifying the schema name, I recommend specifying the schema anyway whether you're using Hue or writing files containing SQL queries, like this: **`prod_schema`**`.dim_us_state_mapping`. Alternatively, you can indicate the schema you want to use by submitting the following code first before submitting your SQL queries:

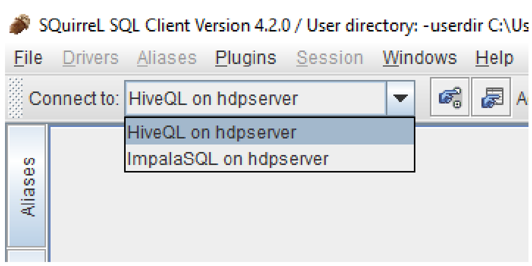
```
USE PROD_SCHEMA;
```

   via HiveQL or ImpalaSQL. If you do neither of those things, your SQL queries will bomb because the default schema is named `default`…and your department's fab tables ain't there, buddy!
3. Occasionally, a table will not appear in the list of tables on the left side. If this happens, click on the Refresh icon just to the right of the plus-sign, ensure the radio button to the left of the text **Clear cache** is selected and then click the Refresh button. If the table still does not show up, try **Perform incremental metadata update** instead. Note that if you have many tables, these options may take some time to complete.
4. If you have thousands of tables in your schema, you may need to ask your Hadoop Administrator to increase the maximum number of tables displayed in Hue.
5. As we discuss below, you can create one or more views to make querying the database easier, especially for people not proficient with SQL joins.

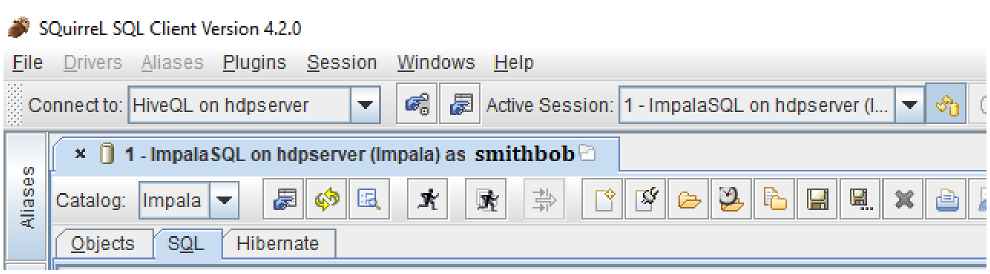
Working with a SQL Client

In this section, we'll briefly show you how to work with SQuirreL. SQL clients tend to be very similar, but the choice of SQuirreL is due to its clean and responsive GUI interface, excellent Help documentation, myriad plug-ins, etc. Okay, okay…who am I kidding?…it's because the name SQuirreL is *adorable*!!

1. To start SQuirreL, double-click on the shortcut on your Desktop.
2. To log into Impala, click the drop-down box to the right of the text **Connect to:** and click **ImpalaSQL on hdpserver**. (Alternatively, you can use the Alias you created from the Aliases pane.)



3. When the **Connect to: ImpalaSQL on hdpserver** dialog box appears, ensure the username/password are correct and then click the Connect button.
4. Once connected, you will see something similar to the following:

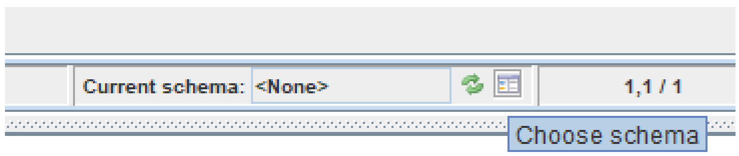


Take note that there's a drop-down box to the right of the text **Connect to:** as well as a drop-down box to the right of the text **Active Session:**. This may lead to some confusion. Whereas the **Connect to:** drop-down box indicates available databases that you are able to connect to, the **Active Session:** drop-down box indicates all of the actively connected to database sessions you have in your current SQuirreL session.

5. Now that you're connected to Impala, you should select your schema. One way is to submit the following SQL code in the SQL window:

```
use prod_schema;
```

Or you can select your schema by clicking on **Choose schema** button at the bottom right of the GUI interface and clicking on your schema. I highly recommend using the `USE` *`schema-name`*`;` syntax or providing the schema name along with the table name: *`schema_name.table-name`*.



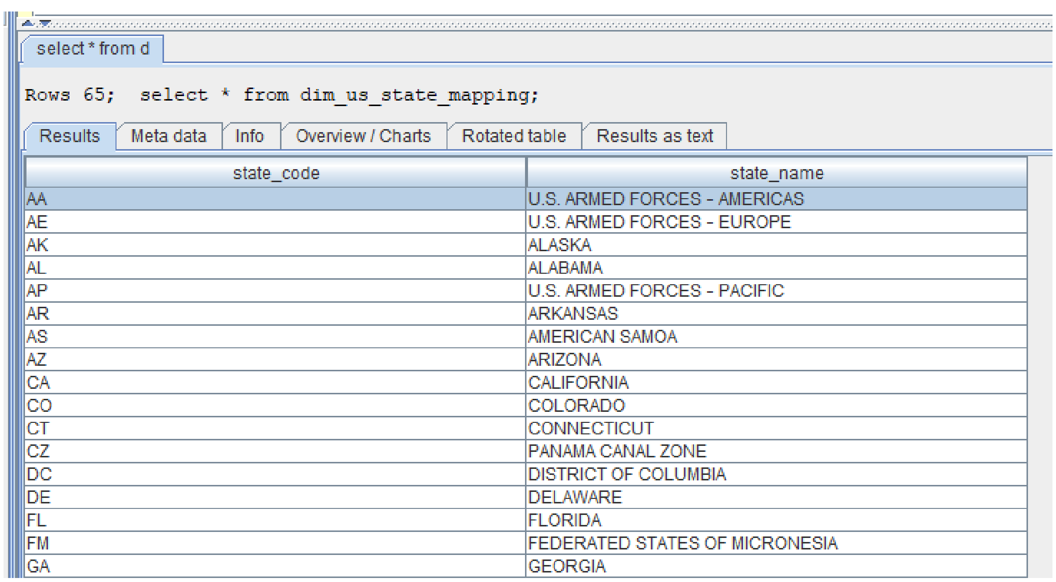
6. To run a SQL query, ensure the SQL tab at the top of the GUI interface is active. Enter your SQL code in the textbox, like this:

```
USE PROD_SCHEMA;
SELECT *
 FROM DIM_US_STATE_MAPPING;
```

or

```
SELECT *
 FROM PROD_SCHEMA.DIM_US_STATE_MAPPING;
```

Next, highlight the code and click the Run SQL button. The results will be displayed in a grid at the bottom of the screen:



state_code	state_name
AA	U.S. ARMED FORCES - AMERICAS
AE	U.S. ARMED FORCES - EUROPE
AK	ALASKA
AL	ALABAMA
AP	U.S. ARMED FORCES - PACIFIC
AR	ARKANSAS
AS	AMERICAN SAMOA
AZ	ARIZONA
CA	CALIFORNIA
CO	COLORADO
CT	CONNECTICUT
CZ	PANAMA CANAL ZONE
DC	DISTRICT OF COLUMBIA
DE	DELAWARE
FL	FLORIDA
FM	FEDERATED STATES OF MICRONESIA
GA	GEORGIA

There are several tabs available for you to peruse:

- Results – A grid containing your SQL query's output.
- Meta data – Column information based on your SQL query.
- Info – Information such as execution time, number of rows returned, etc.

- Overview/Charts – Summary data similar to Microsoft Excel PivotTable and the ability to create charts from the resulting data.
- Rotated table – The same information contained in the Results grid, but pivoted. Nice!!
- Results as text – The same information contained in the Results grid, but in old-fashioned text format, just like the good ol' days!

7. If you would like to export the data, ensure the Results tab is active. Right click over the grid and click the popup menu item **Export CSV / MS Excel / XML / JSON:**. Once the Export dialog box appears, fill in the name of the output file in the **Export to file:** input box, select your export format from the **Export formats** section, select whether you want the headers to appear by checking/unchecking the checkbox to the left of the text **Include column headers**, determine if you want the entire table exported or only a subset of it (**Export complete table** vs **Export selection**) and, finally, click OK to export the data. Depending on the size of your query results, this may take some time to complete.

Along with the SQL tab at the top of the GUI, the Objects tab displays some very important information:

1. Metadata – This tab displays useful properties and values such as getURL, getDriverName, JDBC Driver `CLASSNAME`, JDBC Driver `CLASSPATH` and more.
2. Data Types – This tab displays the data types available for HiveQL (if logged in to Hive) or ImpalaSQL (if logged in to Impala).
3. Numeric Functions – This tab displays the numeric functions available to use in your SQL queries. Note that this may not be a complete list, so please check the documentation.
4. String Functions – This tab displays the string functions available to use in your SQL queries. Note that this may not be a complete list, so please check the documentation.
5. System Functions – This tab displays the system functions available to use in your SQL queries. Note that this may not be a complete list, so please check the documentation.
6. Time/Date Functions – This tab displays the time and date functions available to use in your SQL queries. Note that this may not be a complete list, so please check the documentation.

Finally, for those SQL programmers on your team who won't use the Linux edge node server, an important factor in deciding on a SQL client is the ability to export data from it. When you and your team are deciding on a SQL client to use, this particular feature better be spot on…or there's a-gonna a-be a-trouble, Lucy!

Chapter 8 – The One About ImpalaSQL

In this chapter, we discuss both Data Manipulation Language (DML) as well as Data Definition Language (DDL) for ImpalaSQL. I assume you're somewhat familiar with SQL (if not, please see *Chapter 6 – Introduction to SQL*) , but we'll still explain the syntax below in some detail.

Logging in Using `impala-shell`

As indicated earlier, you can access Impala via a SQL Client GUI, the Hadoop User Experience (Hue) web interface or the Linux command line utility `impala-shell`. In this section, we show you how to access Impala to issue ImpalaSQL commands using the Linux command line utility `impala-shell`. Note that we discuss how to use some `impala-shell` command line switches to pass in parameters to ImpalaSQL queries in *Chapter 21 – Running ImpalaSQL from the Linux Command Line*, but, in this section, we solely discuss how to log into Impala using `impala-shell`.

After logging into the Linux edge node server via PuTTY, depending on how your fantasmagorical Hadoop Admininstrator has set things up, you may be able to simply type in `impala-shell` at the Linux command line to access Impala to issue ImpalaSQL queries:

```
[smithbob@lnxserver ~]$ impala-shell
Starting Impala Shell without Kerberos authentication
Opened TCP connection to lnxserver.com:21000
Connected to lnxserver.com:21000
Server version: impalad version 3.4.0-SNAPSHOT RELEASE (build
68b919fc8a5907648349aa48eefc894e15a5a1a5)
*****************************************************************************
Welcome to the Impala shell.
(Impala Shell v3.4.0-SNAPSHOT (27b919f) built on Tue Aug  3 21:19:39 UTC 2021)

The SET command shows the current value of all shell and query options.
*****************************************************************************
[lnxserver.com:21000] default>
```

Note that the command prompt indicates you've logged into **`lnxserver.com:21000`** in the `default` schema. If there was an issue, you'd see a *not connected* message rather than the host name and port number.

If you'd like to specify exactly which Impala server and port to log into, you can issue the following command instead:

```
[smithbob@lnxserver ~]$ impala-shell --impalad=lnxserver.com:21000
```

Instead of `--impalad=` you can use `-i`:

```
[smithbob@lnxserver ~]$ impala-shell -i lnxserver.com:21000
```

You can log into a specific schema directly by using either the `--database=` or `-d` followed by the name of the schema, such as `prod_schema`:

```
[smithbob@lnxserver ~]$ impala-shell -i lnxserver.com:21000 -d prod_schema
```

To log in as a specific user, you can specify either `--user=` or `-u` followed the name username:

```
[smithbob@lnxserver ~]$ impala-shell -i lnxserver.com:21000
                                     -d prod_schema_export
                                     -u smithbob
```

You may prompted for your password.

If your company uses the Kerberos computer-network authentication protocol, you can specify the `-k` option on the command line and you should be logged in automatically. Please see the responses to the Hadoop Administrator E-Mail in *Chapter 2 – Hadoop Administrator E-Mail* for more on Kerberos.

```
[smithbob@lnxserver ~]$ impala-shell -k -i lnxserver.com:21000 -d prod_schema
```

Please work with your stellar Hadoop Administrator if you're having problems logging in.

Data Manipulation Language (DML)

For the most part, the SQL DML you learned working with your legacy database can be immediately applied to ImpalaSQL with only minor chaffing. In ImpalaSQL, the SQL Data Manipulation Language (DML) syntax looks broadly like the following:

```
WITH Clause
SELECT col1,col2,...
 FROM tbl_name
 WHERE subsetting_conditions
 GROUP BY col1,... [ CUBE() | ROLLUP() | GROUPING SETS() ]
 HAVING post_subsetting_conditions
 ORDER BY col1,...
 LIMIT # OFFSET #
 TABLESAMPLE SYSTEM(percent) REPEATABLE(seed)
```

Since I assume you have some SQL knowledge, I'll briefly describe the syntax above:

- `WITH` Clause – This clause, also known as *Common Table Expressions* or the *Subquery Factoring Clause*, allows you to neaten up your SQL code by creating queries up-front which can then be used in the subsequent SQL code. For example, below we join the tables `dim_us_state_mapping` and `dim_postal_code` by using the `WITH` Clause:

```
WITH vwPC AS (
              SELECT POSTAL_CODE,STATE_CODE
               FROM PROD_SCHEMA.DIM_POSTAL_CODE
               WHERE STATE_CODE IN ('NJ','PA')
             ),
     vwUSM AS (
               SELECT STATE_CODE,STATE_NAME
                FROM PROD_SCHEMA.DIM_US_STATE_MAPPING
                WHERE STATE_CODE IN ('NJ','PA')
              )
SELECT A.POSTAL_CODE,A.STATE_CODE,B.STATE_NAME
 FROM vwPC A INNER JOIN vwUSM B
 ON A.STATE_CODE=B.STATE_CODE;
```

Note that two common table expressions are being created in the SQL code above: `vwPC` and `vwUSM`. These are just two full-blown SQL queries and are referenced on the `FROM` Clause. As you can imagine, for much larger and more complicated SQL queries, the `WITH` Clause can neaten up your code quite a bit!

Note that the second `WITH` Clause query (`vwUSM`, in the example above) can make use of the first `WITH` Clause's query (`vwPC`, in the example above). This is true of any subsequent SQL query when using the `WITH` Clause.

Be aware that when the SQL query completes, the data generated via the `WITH` Clause is no longer available.

- ☐ `SELECT` Clause – This clause allows you to specify a comma-delimited list of columns you want to appear in the query results. Be aware of the following:
  - You can precede the list of columns with the keyword `DISTINCT` to return a de-duplicated list of values based on the selected columns.
  - Short-hand notation for *return all columns* is indicated by an asterisk (*) instead of a comma-delimited list of columns. It's generally a good idea to avoid this usage since specifying a list of columns makes for more lucid SQL code.
  - If you've specified a table alias, such as the letter `A` and `B` in the SQL code above, you can specify `A.*` and/or `B.*` on the `SELECT` Clause to retrieve all columns from the table associated with the alias (but, read the comment above). Note that the repetition of columns will not cause an error when just viewing the query results. This is not true if you're creating a table using the CTAS syntax (see below).
  - You can use the `AS` *`expr-name`* syntax to either rename a column or to give a name to a calculation/expression. This is called a *column alias*. For example,

```
SELECT COUNT(*) AS ROW_COUNTS,5*2 AS TEN
 FROM PROD_SCHEMA.DIM_US_STATE_MAPPING;
```

  - In the example above, we set the calculation `5*2` to the alias `ten`. Note that `5*2` is an example of a numeric literal; that is, a constant value that will not change. You can also use string literals as well as date literals. A string literal, as you can probably guess, is of the form `'text' as alias`. We discuss date and datetime literals later in the book.
  - Unlike other SQL variants, you don't need to specify the `FROM` Clause if you just want to perform a calculation. Recall that Oracle supplies the table `DUAL` which can be used with calculations, but it's not needed in ImpalaSQL:

```
SELECT .25 * 35000;
```

- ☐ `FROM` Clause – This clause allows you to specify one or more comma-delimited list of tables, views, common table expressions, etc. from which to query.
  - If more than one table, view, etc. is specified, then it's a good idea to specify an alias for it. An alias is just short-hand notation and can be as simple as a letter of the alphabet: *`table-name-1`* **`A`**, *`table-name-2`* **`B`**, and so on, but continue to use the method you're comfortable with.
  - If working with only one table, view, etc., then there's no need to use an alias.

- ☐ `WHERE` Clause – This clause allows you to specify subsetting conditions on the incoming data. (To specify subsetting conditions on the outgoing results, see the `HAVING` Clause.).
  - Subsetting conditions make use of the standard arithmetic operators: >, <, >=, <=, =, <>, `!=`. Take note that both **<>** and **`!=`** indicate *not equal to*.
  - When working with strings, you can specify either single- (`'`) or double-quotes (`"`).
  - ImpalaSQL functions can be used as well. We talk more about that below.
  - Both the `IN` and `NOT IN` Operators can be used to specify a comma-delimited list of values, either numeric or string.

```
SELECT *
 FROM PROD_SCHEMA.DIM_POSTAL_CODE
 WHERE STATE_CODE IN ('HI','GU')
       AND LONGITUDE>100;
```

- ☐ `GROUP BY` Clause – This clause allows you to specify one or more columns to be used in the summarization of the data. This can be thought of as similar to a Microsoft Excel PivotTable idea…only all grown up and no longer living in his parents' basement.
  - Note that you should probably specify an aggregate function on the `SELECT` Clause. In the example below, I'm counting the number of zip codes within each US two-letter state code using the `COUNT` aggregate function as well as providing a column alias for it: `NBR_ZIPS`.

```
SELECT STATE_CODE,COUNT(*) AS NBR_ZIPS
 FROM PROD_SCHEMA.DIM_POSTAL_CODE
 GROUP BY STATE_CODE;
```

- ☐ `HAVING` Clause – This clause allows you to specify one or more subsetting conditions that apply **after** the SQL query has completed. Recall that the `WHERE` Clause subsets **incoming** table data. For example, in the SQL code below I'm limiting the number of outgoing rows to those states with more than 1000 zip codes:

```
SELECT STATE_CODE,COUNT(*) AS NBR_ZIPS
 FROM PROD_SCHEMA.DIM_POSTAL_CODE
 GROUP BY STATE_CODE
 HAVING COUNT(*) >= 1000;
```

Take note that you cannot use the column alias name `nbr_zips` on the `HAVING` Clause which is why I've specified `COUNT(*)` in the SQL code above.

- ☐ `ORDER BY` Clause – This clause allows you to specify one or more columns used to order the results of the SQL query.
  - Don't specify an `ORDER BY` Clause when creating a table because ImpalaSQL will laugh hysterically at you and ignore it anyway…just like most SQL variants.
  - You can use calculations/expressions on this clause. For example, if you want to order the results randomly, you can use the SQL code `ORDER BY RAND()`, or equivalent.
  - By default, sorting is ascending except when the `DESC` keyword is used.
  - You can consider `NULL` as the largest value and, by default, it sorts to the **bottom** when using ascending sorting, but sorts to the **top** when using descending (`DESC`) sorting. You can change this by specifying `NULLS FIRST` or `NULLS LAST` on the `ORDER BY` Clause. In the example below, the `NULL` value(s) will sort to the bottom:

```
SELECT COL1
 FROM (
       SELECT 1 AS COL1
       UNION
       SELECT 2 AS COL1
       UNION
       SELECT NULL AS COL1
      ) A
 ORDER BY COL1 DESC NULLS LAST;

            +------+
            | col1 |
            +------+
            | 2    |
            | 1    |
            | NULL |
            +------+
```

- ☐ `LIMIT`/`OFFSET` Clause – These clauses allows you to limit the total number of rows which appear in the output as well as specify what row offset you want to start with.
  - Note that the syntax is `LIMIT # OFFSET #` in ImpalaSQL,.
  - If you want to make use of `OFFSET`, you must specify an `ORDER BY` Clause to enforce an order on the rows.
  - If you leave off the `OFFSET`, you will be limiting the number of rows output. In the example below, you will only see the first 10 rows of the query results:

```
SELECT *
 FROM PROD_SCHEMA.DIM_POSTAL_CODE
 WHERE STATE_CODE IN ('HI','GU')
       AND LONGITUDE>100
 ORDER BY POSTAL_CODE
 LIMIT 10;
```

```
+-------------+---------------+------------+----------+-----------+
| postal_code | city          | state_code | latitude | longitude |
+-------------+---------------+------------+----------+-----------+
| 96910       | HAGATNA       | GU         | 13.47    | 144.74    |
| 96912       | DEDEDO        | GU         | 13.51    | 144.83    |
| 96913       | BARRIGADA     | GU         | 13.46    | 144.79    |
| 96915       | SANTA RITA    | GU         | 13.38    | 144.66    |
| 96916       | MERIZO        | GU         | 13.26    | 144.66    |
| 96917       | INARAJAN      | GU         | 13.27    | 144.74    |
| 96919       | AGANA HEIGHTS | GU         | 13.46    | 144.74    |
| 96921       | BARRIGADA     | GU         | 13.46    | 144.79    |
| 96923       | MANGILAO      | GU         | 13.43    | 144.79    |
| 96928       | AGAT          | GU         | 13.38    | 144.65    |
+-------------+---------------+------------+----------+-----------+
```

- By default, the first row is `OFFSET 0`, not `OFFSET 1`. In the example below, we order by `postal_code`, offset by 5 and limit to 10 rows:

```
SELECT *
 FROM PROD_SCHEMA.DIM_POSTAL_CODE
 WHERE STATE_CODE IN ('HI','GU')
       AND LONGITUDE>100
 ORDER BY POSTAL_CODE
 LIMIT 10
 OFFSET 5;
```

```
+-------------+---------------+------------+----------+-----------+
| postal_code | city          | state_code | latitude | longitude |
+-------------+---------------+------------+----------+-----------+
| 96917       | INARAJAN      | GU         | 13.27    | 144.74    |
| 96919       | AGANA HEIGHTS | GU         | 13.46    | 144.74    |
| 96921       | BARRIGADA     | GU         | 13.46    | 144.79    |
| 96923       | MANGILAO      | GU         | 13.43    | 144.79    |
| 96928       | AGAT          | GU         | 13.38    | 144.65    |
| 96929       | YIGO          | GU         | 13.53    | 144.88    |
| 96931       | TAMUNING      | GU         | 13.48    | 144.77    |
| 96932       | HAGATNA       | GU         | 13.47    | 144.74    |
+-------------+---------------+------------+----------+-----------+
```

  Note that my `WHERE` Clause limits the query to a total of 13 rows, but the `OFFSET` starts at row 6, so only 8 rows are returned in total.
- When using the `ORDER BY` Clause in a subquery (see the section below), you must provide a `LIMIT` Clause or the query will bomb. You don't need to sort tables when merging because the database handles that. But, you may need to sort the table for certain functions, like `GROUP_CONCAT`, in order to obtain the desired results.

☐ `TABLESAMPLE` Clause – This clause allows you to sample a percentage of the data from a non-`KUDU` table. This is a slight lie because the percentage you specify in the `SYSTEM` option may retrieve more rows than you expect since retrieval is based on the underlying number of data files and not the percentage of the number of rows.

For example, the table `PROD_SCHEMA.DIM_US_STATE_MAPPING` contains `65` rows whereas the table `PROD_SCHEMA.DIM_POSTAL_CODE` contains `43,689` rows. Let's use the `TABLESAMPLE` Clause to pull in `10%` of the rows of each table:

```
SELECT COUNT(*) AS TOTAL_ROWS
 FROM DIM_POSTAL_CODE;

+------------+
| total_rows |
+------------+
| 43689      |
+------------+

SELECT COUNT(*) AS SAMPLE_ROWS
 FROM DIM_POSTAL_CODE
 TABLESAMPLE SYSTEM(10);

+-------------+
| sample_rows |
+-------------+
| 43689       |
+-------------+

SELECT COUNT(*) AS TOTAL_ROWS
 FROM DIM_US_STATE_MAPPING;

+------------+
| total_rows |
+------------+
| 65         |
+------------+

SELECT COUNT(*) AS TOTAL_ROWS
 FROM DIM_US_STATE_MAPPING
 TABLESAMPLE SYSTEM(10);

+------------+
| total_rows |
+------------+
| 7          |
+------------+
```

Well!! That's weird, isn't it!?! You would think that about `4368`-ish rows would be returned from `DIM_POSTAL_CODE`, but the entire table is returned! *Hi-ya!!* But, everything works fine for the table `DIM_US_STATE_MAPPING`. This occurs because the data in the table `DIM_POSTAL_CODE` resides in only one file in HDFS, whereas the data in the table `DIM_US_STATE_MAPPING` resides in `65` separate files in HDFS. To see this, you can issue `SHOW FILES IN` *`table_name`*`;` at the `impala-shell` command line to display the underlying files in HDFS for the table (output diddled with to save space):

```
SHOW FILES IN DIM_POSTAL_CODE;
+-----------------------------------------------------------------------------------------------------------+--------+
| Path                                                                                                      | Size   |
+-----------------------------------------------------------------------------------------------------------+--------+
| ...snip.../dim_postal_code/delta_1_1/534a24bb518304f4-4750ea1b00000000_273855379_data.0.parq              | 1.28MB |
+-----------------------------------------------------------------------------------------------------------+--------+

SHOW FILES IN DIM_US_STATE_MAPPING;
+-----------------------------------------------------------------------------------------------------------+--------+
| Path                                                                                                      | Size   |
+-----------------------------------------------------------------------------------------------------------+--------+
| ...snip.../dim_us_state_mapping/delta_10_10/cb43aac629622559-3e83bb3e00000000_1993903573_data.0.parq | 643B   |
| ...snip.../dim_us_state_mapping/delta_11_11/dc4a57323621dc6f-c63ac57b00000000_2051461713_data.0.parq | 664B   |
...snip...
```

```
| ...snip.../dim_us_state_mapping/delta_12_12/6442b559135f45ac-4526597b00000000_693829157_data.0.parq  | 708B   |
| ...snip.../dim_us_state_mapping/delta_13_13/b34bd5a18f3edb85-e4cf27e600000000_716542679_data.0.parq  | 729B   |
| ...snip.../dim_us_state_mapping/delta_9_9/a14856e144701a23-1b4bd2e200000000_1973402835_data.0.parq   | 657B   |
+----------------------------------------------------------------------------------------------------+--------+
```

In order to ensure that the sampled data is exactly the same each time you run the SQL query, specify the REPEATABLE keyword along with an integer seed value. For example, the following SQL query will always return the same rows (up to system changes such as adding new data, re-computing statistics, etc.):

```
SELECT *
 FROM DIM_US_STATE_MAPPING
 TABLESAMPLE SYSTEM(10) REPEATABLE(31415)
 ORDER BY STATE_CODE;

+------------+--------------------------+
| state_code | state_name               |
+------------+--------------------------+
| CO         | COLORADO                 |
| KY         | KENTUCKY                 |
| LA         | LOUISIANA                |
| MH         | MARSHALL ISLANDS         |
| MI         | MICHIGAN                 |
| MP         | NORTHERN MARIANA ISLANDS |
| SD         | SOUTH DAKOTA             |
+------------+--------------------------+
```

- Subqueries – Rather than using the crisp-and-clean WITH Clause, you can use *flabby-and-filthy* subqueries on the FROM Clause by surrounding each SQL query with parentheses. Note that you must specify an alias for each subquery or the entire query will bomb like telling a joke at a funeral. For example, we can re-write the SQL query using the WITH Clause above using subqueries instead:

```
SELECT A.POSTAL_CODE,A.STATE_CODE,B.STATE_NAME
 FROM (
       SELECT POSTAL_CODE,STATE_CODE
        FROM PROD_SCHEMA.DIM_POSTAL_CODE                <----- FLABBY!
        WHERE STATE_CODE IN ('NJ','PA')
      ) A INNER JOIN (
                      SELECT STATE_CODE,STATE_NAME
 FILTHY! ----->        FROM PROD_SCHEMA.DIM_US_STATE_MAPPING
                       WHERE STATE_CODE IN ('NJ','PA')
                     ) B
 ON A.STATE_CODE=B.STATE_CODE;
```

As has been demonstrated several times already, subqueries can also be used with IN and NOT IN operators, for example:

```
SELECT STATE_CODE,POSTAL_CODE
 FROM PROD_SCHEMA.DIM_POSTAL_CODE
 WHERE STATE_CODE IN (
                      SELECT STATE_CODE
                       FROM DIM_US_STATE_MAPPING
                       WHERE SUBSTR(STATE_CODE,1,1)='P'
                     );
```

If NULLs appear in your data – and when don't the little bastards? – ensure that your query is functioning properly when using both IN and NOT IN.

- Joins/ON Clause – There are several options for joining tables using ImpalaSQL. With the exception of a Cartesian Product, you must specify the ON Clause to indicate how the tables are to be joined together. Please avoid using a WHERE Clause to specify join criteria. Think of it this way: The WHERE Clause is used

to provided subsetting criteria and the `ON` Clause is used to provider join criteria…and never the twain shall meet!

- *`l-table`* `L INNER JOIN` *`r-table`* `R` – Use `INNER JOIN` to indicate you want to keep only the matching rows from both *`l-table`* and *`r-table`*. In the example above, we're joining the two subqueries using an `INNER JOIN` and specifying the `ON` Clause where the state code matches both tables. Note that you can leave off `INNER` and just specify `JOIN` to indicate an `INNER JOIN`, but please don't do that…it makes me sad!
- *`l-table`* `L LEFT JOIN` *`r-table`* `R` – A `LEFT JOIN` is just an `INNER JOIN` with the addition of the unmatched rows from the *`l-table`* tacked on to the output.
- *`l-table`* `L RIGHT JOIN` *`r-table`* `R` – A `RIGHT JOIN` is just an `INNER JOIN` with the addition of the unmatched rows from the *`r-table`* tacked on to the output.
- *`l-table`* `L FULL JOIN` *`r-table`* `R` – A `FULL JOIN` is just an `INNER JOIN` with the addition of the unmatched rows from the *`l-table`* as well as the unmatched rows from the *`r-table`* tacked on to the output.
- Note that you can also specify the keyword `OUTER`: `LEFT OUTER JOIN`, `RIGHT OUTER JOIN`, and `FULL OUTER JOIN`. Since we programmers are paid based on the number of characters in our SQL queries, best to add `OUTER`!
- Cartesian Product – In other databases, a Cartesian Product can be specified using a comma between the two tables on the `FROM` Clause. ImpalaSQL doesn't allow for this syntax, but instead provides the `CROSS JOIN` keyword to accomplish the same thing. For example, the following will NOT work and is complete and utter crap:

```
SELECT COUNT(*)
 FROM PROD_SCHEMA.DIM_US_STATE_MAPPING,
                                  PROD_SCHEMA.DIM_US_STATE_MAPPING;
```

  But, the following will work and is not crap:

```
SELECT COUNT(*)
 FROM PROD_SCHEMA.DIM_US_STATE_MAPPING CROSS JOIN
                                  PROD_SCHEMA.DIM_US_STATE_MAPPING;
```

  Note that you can include a `WHERE` Clause to subset the incoming data as well as a `HAVING` Clause to subset the outgoing data.

- `USING` Clause – If the join column(s) are spelled the same in both the left and right tables, you can skip the `ON` Clause and use the `USING` Clause instead. For example, in the code below, we replace the `ON` Clause `ON A.STATE_CODE=B.STATE_CODE` with `USING (STATE_CODE)` since the column `STATE_CODE` is spelled the same in both subqueries:

```
WITH vwPC AS (
              SELECT POSTAL_CODE,STATE_CODE
               FROM PROD_SCHEMA.DIM_POSTAL_CODE
               WHERE STATE_CODE IN ('NJ','PA')
             ),
     vwUSM AS (
               SELECT STATE_CODE,STATE_NAME
                FROM PROD_SCHEMA.DIM_US_STATE_MAPPING
                WHERE STATE_CODE IN ('NJ','PA')
              )
SELECT A.POSTAL_CODE,A.STATE_CODE,B.STATE_NAME
 FROM vwPC A INNER JOIN vwUSM B
 USING (STATE_CODE);
```

- `SEMI JOIN` Clause – It would be nice, and would certainly make life easier, if every table we worked with had only one row per the join criteria, but that's not always the case (sadly). Because of this, when joining tables together, there's the possibility of a data-related nuclear explosion if one or both tables contain many rows on the join columns. You can use a semi-join to prevent this type

of data explosion. A LEFT SEMI JOIN indicates you want the rows from the *l-table* returned, and no duplicates, based on the matching ON Clause criteria. The RIGHT SEMI JOIN Clause is similar to LEFT SEMI JOIN, but reversed for the table on the right, *r-table*.

- ANTI JOIN Clause – When using a LEFT OUTER JOIN, only those rows matching the *r-table* are output. If you would like those rows NOT matching the *r-table* returned, you can use a LEFT ANTI JOIN instead. A similar concept follows when using RIGHT ANTI JOIN.

☐ UNION and UNION ALL Clauses – You can append the output from two or more complete SQL queries by using the UNION or UNION ALL clauses. If you specify UNION, the data is de-duplicated based on all of the columns. If you specify UNION ALL instead, the data is **not** de-duplicated and the results from one query is thrown under the results from the other query. For example, let's append, **without de-deduplication**, the zip codes from both New Jersey and Pennsylvania:

```
WITH vwNJ AS (
              SELECT POSTAL_CODE
               FROM PROD_SCHEMA.DIM_POSTAL_CODE
               WHERE STATE_CODE='NJ'
             ),
     vwPA AS (
              SELECT POSTAL_CODE
               FROM PROD_SCHEMA.DIM_POSTAL_CODE
               WHERE STATE_CODE='PA'
             )
SELECT POSTAL_CODE
 FROM vwNJ
UNION ALL
SELECT POSTAL_CODE
 FROM vwPA;
```

☐ Operators – There are several operators which can be used in your SQL code.

- LIKE and ILIKE Operators – These operators can be used in a WHERE Clause to subset the incoming data. The operator ILIKE is the case-insensitive version of the LIKE operator. They both make use of a string template containing one or more of the following symbols with special meaning:
  - Percent Sign (%) – This symbol indicates one or more characters are to appear at a specific point in the template.
  - Underscore (_) – This symbol indicates exactly one character appears at a specific point in the template.

  For example, below we search the PROD_SCHEMA.DIM_US_STATE_MAPPING table for state names containing the text ARMED followed by the text FORCES:

```
SELECT * FROM
 PROD_SCHEMA.DIM_US_STATE_MAPPING
 WHERE STATE_NAME LIKE '%ARMED%FORCES%';

+------------+------------------------------+
| state_code | state_name                   |
+------------+------------------------------+
| AA         | U.S. ARMED FORCES - AMERICAS |
| AE         | U.S. ARMED FORCES - EUROPE   |
| AP         | U.S. ARMED FORCES - PACIFIC  |
+------------+------------------------------+
```

- IN and NOT IN Operators – As we've seen already, these operators allow you to specify a comma-delimited list of items you either want or don't want pulled from the SQL query results. Note that it might be tempting to include a butt-load of comma-delimited items when using these operators, but there's a point you might just want to create a table containing these items and then perform an

INNER JOIN instead. You know you've gotten to that point if you have to use the page down button several times to get to the end of your SQL query.

```
SELECT POSTAL_CODE,STATE_CODE
 FROM PROD_SCHEMA.DIM_POSTAL_CODE
 WHERE STATE_CODE IN ('NJ','PA')
```

- EXISTS and NOT EXISTS Operators – As we indicated earlier, the keywords INTERSECT, MINUS and EXCEPT are not available in ImpalaSQL, as yet. You can code around the missing INTERSECT keyword, for example, by performing an appropriate INNER JOIN or other SQL query. Both MINUS and EXCEPT (they are cinnamons…er…synonyms), can be coded using the NOT EXISTS Operator on the WHERE Clause with a correlated subquery.

```
SELECT A.*
 FROM PROD_SCHEMA.DIM_POSTAL_CODE A
 WHERE NOT EXISTS (
                   SELECT B.*
                    FROM B
                    WHERE A.STATE_CODE=B.STATE_CODE
                  )
```

- BETWEEN/AND Operator – Rather than specifying a range of values using the arithmetic operators along with the AND operator, you can perform a similar task using BETWEEN/AND. Take note that the value provided after the keyword BETWEEN is tested for *greater than or equal to* whereas the value provided after the keyword AND is tested for *less than or equal to*. For example, in the code below, the value 20 is the upper bound and 20.1, say, will not be included. The SQL queries below return the same results.

```
SELECT STATE_CODE,COUNT(*) AS NBR_ZIPS
 FROM PROD_SCHEMA.DIM_POSTAL_CODE
 GROUP BY STATE_CODE
 HAVING COUNT(*) >= 10 AND COUNT(*) <= 20;
```

…is equivalent to…

```
SELECT STATE_CODE,COUNT(*) AS NBR_ZIPS
 FROM PROD_SCHEMA.DIM_POSTAL_CODE
 GROUP BY STATE_CODE
 HAVING COUNT(*) BETWEEN 10
                     AND 20;
```

☐ CASE Expression – Occasionally, you may need to recode a column to something more relevant for a particular analysis. For example, let's create a column called CULTURE which will be set to Hollywood when the zip code is 90027, Broadway when the zip code is 10018, and No Culture everywhere else:

```
SELECT STATE_CODE,POSTAL_CODE,
       CASE
        WHEN POSTAL_CODE='90027' THEN 'Hollywood'
        WHEN POSTAL_CODE='10018' THEN 'Broadway'
        ELSE                          'No Culture'
       END AS CULTURE
 FROM PROD_SCHEMA.DIM_POSTAL_CODE;
```

Note that the keyword END ends the CASE Expression. As you see above, you can use AS *column-name* to name the new column (AS CULTURE, in the code above). You can use the CASE Expression with a GROUP BY Statement along with an aggregate function, but be sure to exclude AS *column-name* on the GROUP BY:

```
SELECT CASE
```

```
         WHEN POSTAL_CODE='90027' THEN 'Hollywood'
         WHEN POSTAL_CODE='10018' THEN 'Broadway'
         ELSE                          'No Culture'
        END AS CULTURE,
        COUNT(*) AS ROW_CNTS
FROM PROD_SCHEMA.DIM_POSTAL_CODE
GROUP BY CASE
            WHEN POSTAL_CODE='90027' THEN 'Hollywood'
            WHEN POSTAL_CODE='10018' THEN 'Broadway'
            ELSE                          'No Culture'
           END;
```

Data Definition Language (DDL)

Similar to Data Manipulation Language (DML), the SQL Data Definition Language (DDL), such as `CREATE TABLE`, `INSERT INTO`, etc., you used with your legacy database can be *almost* immediately applied to ImpalaSQL. As we discussed in *Chapter 4 – A Teensy-Weensy Chat about Hadoop*, there's the issue of the storage formats as well as concepts such as creating tables, partitioning, computing statistics, etc. which need to be described. Let's start off simple with the data types and then we'll move on to the DDL statements.

Note that in this chapter, we poo-poo and eschew discussions about partitioning and other performance improvements and pick that up in *Chapter 16 – SQL Performance Improvements*. It's just too much to throw at you in one swell foop.

Also, be aware that certain DDL statements work with certain storage formats. I know! Weird, huh? For example, a table created using the `KUDU` storage format cannot use the `TRUNCATE` Statement, but you can use a `DELETE` Statement without a `WHERE` Clause as a high-class substitute. We indicate these discrepancies below.

ImpalaSQL Data Types

Before we discuss DDL, let's take a tour of the data types available in ImpalaSQL:

- ☐ **Integral Data Types** – The following data types are used when integral values with various ranges are needed such as for identifiers, counts, and so on. To save storage space, select the data type with the smallest range.
  - `TINYINT` – This data type can hold a range of values from `-128` to `+127` and uses one byte of storage.
  - `SMALLINT` – This data type can hold a range of values from `-32768` to `+32767` and uses two bytes of storage.
  - `INT` – This data type can hold a range of values from `-2,147,483,648` to `+2,147,483,647` and uses four bytes of storage.
  - `BIGINT` – This data type can hold a range of values from `-9,223,372,036,854,775,808` to `+9,223,372,036,854,775,807` and uses eight bytes of storage. (This data type is also known as a *daaaaayyyyyuuuuummmmm!*)

- ☐ **Floating Point Data Types** – The following data types are used when single- or double-point floating values are needed such as for latitudes, longitudes, the air speed velocity of an unladen swallow, etc. Note that these data types are IEEE 754 Single/Double Precision numbers.
  - `FLOAT` – This data type is a single-precision floating point number ranging from `-1.40E-45` to `+3.40E+38` and uses four bytes of storage.
  - `DOUBLE` – This data type is a double-precision floating point number ranging from `-4.94E-324` to `+1.79E+308` and uses eight bytes of storage.
  - `REAL` – This is an alias for `DOUBLE`.

- **Decimal Data Type** – This data type is used when both FLOAT and DOUBLE are way too over-the-top, such as when storing currency values. Unlike the ranges presented above, the DECIMAL data type has the same range regardless if the value is negative or positive.
  - DECIMAL(*precision*,*scale*)
    - *precision* – This value indicates the total number of digits in the number regardless if it's to the left of the decimal place or right. This value can range from 1 to 38.
    - *scale* – This value indicates the number of digits for the fractional part only. This value must be less than or equal to the precision.
  - DECIMAL – This variation is the same as DECIMAL(9,0) and ranges from -999,999,999 to +999,999,999 with no decimal places.
  - DECIMAL(38,0) – This is the largest value the decimal data type can attain with 38 total digits and no decimal places.
  - DECIMAL(38,38) – This is the most precise value the decimal data type can attain with 38 total digits after the decimal place. Not useful for currency (except maybe crypto! HA!), but if you enjoy weighing electrons on your bathroom scale in your spare time…
  - DECIMAL(8,2) – As an example, this variation will store six digits to the left of the decimal place and two digits to the right of the decimal place for a total range of -999,999.99 to +999,999.99.

- **Text-Related Data Types** – These data types are used to store text.
  - STRING – This data type stores approximately 1,000,000,000 (1GB) characters maximum. I prefer this data type over both CHAR and VARCHAR since it eliminates the need to specify the maximum number of allowable characters.
  - CHAR(*max_length*) – This data type stores fixed-length text where *max_length* ranges from 1 to 255. Note that text inserted into this column with less than the *max_length* number of characters will be padded with blanks. I'd probably avoid this data type in favor of the STRING data type.
  - VARCHAR(*max_length*) – This data type stores fixed-length text where *max_length* ranges from 1 to 65,535. I'd probably avoid this data type in favor of the STRING data type.

- **Boolean Data Type** – This data type is useful when you need to store only two choices.
  - BOOLEAN – This data type stores only two values: true or false.

- **Date/Time Data Types** – These data types are used to stores dates and times. Since working with dates and times can be a pain, see *Chapter 10 – Voyage of the Damned (Dates & Times – ImpalaSQL Edition)* for more detailed information.
  - DATE – This data type stores a date ranging from 0001-01-01 to 9999-12-31.
  - TIMESTAMP – This data type stores date/times ranging from breakfast on 1400-01-01 to late night snack on 9999-12-31.

CREATE TABLE Statement (TEXTFILE/PARQUET)

Occasionally, you'll find that you need to create a table and ImpalaSQL turns your dreams into reality. The CREATE TABLE Statement shown below applies to both the TEXTFILE and PARQUET storage formats, not the KUDU storage format. We discuss CREATE TABLE as it applies to the KUDU storage format later on in the chapter. Similar to the SELECT Statement and its associated DML buddies, enter the DDL commands in the order shown below. Note that the commands with a grey background are optional.

```
CREATE EXTERNAL TABLE database_name.table_name
 (
  column_name_1 data_type_1 COMMENT 'column comment 1',
  column_name_2 data_type_2 COMMENT 'column comment 2',
  ...
  column_name_n data_type_n COMMENT 'column comment n'
 )
 PARTITIONED BY (
                 column_name_p1 data_type_p1 COMMENT 'column comment p1',
```

```
                 column_name_p2 data_type_p2 COMMENT 'column comment p2',
                 ...
                 column_name_pk data_type_pk COMMENT 'column comment pk'
                )
SORT BY (column_name_i, column_name_j, ...)
COMMENT 'table-comment'
ROW FORMAT row-format
WITH SERDEPROPERTIES (
                       'key-1','value-1',
                       'key-2','value-2',
                       ...
                       'key-m','value-m'
                      )
STORED AS storage-format
LOCATION 'HDFS-path-to-data-file-directory'
CACHED IN 'cache-pool-name'
 WITH REPLICATION = replication-value | UNCACHED
TBLPROPERTIES (
                'key-1','value-1',
                'key-2','value-2',
                ...
                'key-r','value-r'
               )
;
```

At its very simplest, you can create a new table with a bare minimum set of keywords:

```
CREATE TABLE PROD_SCHEMA.DIM_POSTAL_CODE(
                                         POSTAL_CODE STRING,
                                         CITY        STRING,
                                         STATE_CODE  STRING,
                                         LATITUDE    DOUBLE,
                                         LONGITUDE   DOUBLE
                                        );
```

But, the code above assumes, among other things, you want your table to be managed, use no partitioning scheme, and have a storage format of `TEXTFILE`. If you'd prefer to use the `PARQUET` storage format, you must include the `STORED AS` Clause:

```
CREATE TABLE PROD_SCHEMA.DIM_POSTAL_CODE(
                                         POSTAL_CODE STRING,
                                         CITY        STRING,
                                         STATE_CODE  STRING,
                                         LATITUDE    DOUBLE,
                                         LONGITUDE   DOUBLE
                                        )
 STORED AS PARQUET;
```

If you'd like to add table as well as column comments, include the `COMMENT` Clause:

```
USE PROD_SCHEMA;                                              Column comments
CREATE TABLE DIM_POSTAL_CODE(
                             POSTAL_CODE STRING COMMENT '5-DIGIT POSTAL CODE',
                             CITY        STRING COMMENT 'CITY NAME',
                             STATE_CODE  STRING COMMENT '2-LETTER STATE CODE',
                             LATITUDE    DOUBLE COMMENT 'LATITUDE',
                             LONGITUDE   DOUBLE COMMENT 'LONGITUDE'
                             )
 COMMENT 'UNITED STATES POSTAL CODE TABLE'  <-- Table comment
```

```
 STORED AS PARQUET;
```

Note the table and column comments will appear when issuing a describe on the table:

```
+-------------+--------+---------------------+
| name        | type   | comment             |
+-------------+--------+---------------------+
| postal_code | string | 5-DIGIT POSTAL CODE |
| city        | string | CITY NAME           |
| state_code  | string | 2-LETTER STATE CODE |
| latitude    | double | LATITUDE            |
| longitude   | double | LONGITUDE           |
+-------------+--------+---------------------+
```

We talk about external tables later on in this chapter.

CREATE TABLE AS Statement (TEXTFILE/PARQUET)

If you'd like to create a new table based on a SQL query, you can use the CREATE TABLE AS (CTAS) Statement instead of creating the table up-front and using the INSERT Statement to jam data into it:

```
CREATE EXTERNAL TABLE database_name.table_name
 PARTITIONED BY (column_name_p1,column_name_p2...column_name_pk)
 SORT BY (column_name_i, column_name_j, ...)
 COMMENT 'table-comment'
 ROW FORMAT row-format
 WITH SERDEPROPERTIES (
                       'key-1','value-1',
                       'key-2','value-2',
                       ...
                       'key-m','value-m'
                      )
 STORED AS storage-format
 LOCATION 'HDFS-path-to-data-file-directory'
 CACHED IN 'cache-pool-name'
  WITH REPLICATION = replication-value | UNCACHED
 TBLPROPERTIES (
                'key-1','value-1',
                'key-2','value-2',
                ...
                'key-r','value-r'
               )
AS
 select-statement
;
```

At its very simplest, you can create a new (managed) table based on a SQL query like this:

```
CREATE TABLE PROD_SCHEMA.DIM_POSTAL_CODE_BKUP AS
 SELECT *
  FROM PROD_SCHEMA.DIM_POSTAL_CODE;
```

Since the STORED AS Clause was left off, the storage format defaults to TEXTFILE, but you can override this using the STORED AS Clause:

```
CREATE TABLE PROD_SCHEMA.DIM_POSTAL_CODE_BKUP STORED AS PARQUET AS
 SELECT *
  FROM PROD_SCHEMA.DIM_POSTAL_CODE;
```

If you'd like to create a new table based on the definition of an existing table, but without the data, you can use a `WHERE` Clause with a false condition:

```
CREATE TABLE PROD_SCHEMA.DIM_POSTAL_CODE_BKUP AS
 SELECT *
  FROM PROD_SCHEMA.DIM_POSTAL_CODE
  WHERE FALSE;
```

DROP TABLE Statement (TEXTFILE/PARQUET/KUDU)

You can drop a table by using the `DROP TABLE` Statement:

```
DROP TABLE IF EXISTS database_name.table_name PURGE;
```

The optional `IF EXISTS` prevents you from receiving an error message if the table doesn't actually exist. The `PURGE` option will prevent the table from being placed in the recycle bin and space will be freed up immediately.

Be aware of the following caveats, though:

1. **Managed Tables** – When you drop a **managed** table, the table as well as the underlying data will be removed from the database completely. This is probably the behavior you expect to happen, especially from working with your legacy database.
2. **External Tables** – When you drop an **external** table, the table will be removed, but the underlying data will still remain on disk in HDFS. You can override this behavior by providing the table property `external.table.purge` and setting it to `true` in the `CREATE EXTERNAL TABLE` statement

   ```
   CREATE EXTERNAL TABLE PROD_SCHEMA.DIM_POSTAL_CODE(
                                                    POSTAL_CODE STRING,
                                                    CITY        STRING,
                                                    STATE_CODE  STRING,
                                                    LATITUDE    DOUBLE,
                                                    LONGITUDE   DOUBLE
                                                   )
    STORED AS PARQUET
    TBLPROPERTIES('external.table.purge'='true');
   ```

   Now, when you drop this external table, the table **as well as the underlying data** will be vaporized. As you can well imagine, this property can get you into a lotta trouble real fast…like gambling or drinking.

Dropping a table can be especially risky if the `LOCATION` Clause has been specified incorrectly. For example, incorrectly specifying a *parent directory* rather than the table's *subdirectory* will cause all data under the *parent directory* to be blown away taking all subdirectories with it. This is definitely not something you want to happen! If it does, when your Human Resources Department gets done with you, you'll look like something long since forgotten found at the bottom of a deep-fat fryer. EXTREME CAUTION IS ADVISED!! In all seriousness, if this does occur, contact your savior-like Hadoop Administrator IMMEDIATELY and hopefully he/she can pull everything back from the recycle bin.

INSERT Statement (TEXTFILE/PARQUET/KUDU)

You can insert rows of data into a table created with the `TEXTFILE`, `PARQUET` and `KUDU` storage formats. At its very simplest, you can insert rows into a table from rows of another table with or without a subsetting `WHERE` Clause:

```
INSERT INTO PROD_SCHEMA.DIM_POSTAL_CODE
 SELECT *
  FROM PROD_SCHEMA.MISSING_POSTAL_CODES;
```

If you'd prefer to **overwrite** the data in your table rather than just insert additional data into it, you can use the `OVERWRITE` Keyword instead of the `INTO` Keyword:

```
INSERT OVERWRITE PROD_SCHEMA.DIM_POSTAL_CODE
 SELECT *
  FROM PROD_SCHEMA.DIM_POSTAL_CODE_NOT_CRAP_LIKE_THE_OTHER_TABLE;
```

Note that the `OVERWRITE` Keyword cannot be used with a table created with the snooty `KUDU` storage format.

You can also use the `VALUES` Clause to append additional rows of data to a table:

```
INSERT INTO PROD_SCHEMA.DIM_POSTAL_CODE VALUES('99997','???','??',NULL,NULL);
INSERT INTO PROD_SCHEMA.DIM_POSTAL_CODE VALUES('99998','???','??',NULL,NULL);
INSERT INTO PROD_SCHEMA.DIM_POSTAL_CODE VALUES('99999','???','??',NULL,NULL);
```

In general, the syntax for the `INSERT` Statement is as follows:

```
INSERT INTO|OVERWRITE database_name.table_name_1(column-1,
                                                 column-2,
                                                 ...,
                                                 column-n)
 SELECT column-1,column-2,...,column-n
  FROM database_name.table_name_2
  WHERE where-condition
;
```

And the form of the `INSERT` Statement with the `VALUES` Clause is as follows:

```
INSERT INTO|OVERWRITE database_name.table_name(column-1,
                                               column-2,
                                               ...,
                                               column-n)
 VALUES(value-1,value-2,...,value-n);
```

Note that inserting data into a table repeatedly using this form of the `INSERT` Statement with the `VALUES` Clause is very slow, and has other consequences as well. We discuss how to speed up `INSERT`s, as well as much more, in *Chapter 16 – SQL Performance Improvements*.

TRUNCATE TABLE (TEXTFILE/PARQUET)

Less drastic than `DROP TABLE`, you can remove all of the data within a table by using `TRUNCATE TABLE`:

```
TRUNCATE TABLE IF EXISTS database_name.table_name;
```

Note that you cannot use `TRUNCATE TABLE` with tables created with the snooty `KUDU` storage format, but you can use `DELETE` instead (described below).

For example, to remove all of the rows in the `DIM_POSTAL_CODE` table, you can do this:

```
TRUNCATE TABLE PROD_SCHEMA.DIM_POSTAL_CODE;
```

Using Primary Keys and Indexes (TEXTFILE/PARQUET)

You'll be simultaneously surprised, stunned, enthralled and shocked to know that there's no `CREATE INDEX` Statement and no way to indicate a primary key in ImpalaSQL for both the `TEXTFILE` and `PARQUET` storage

formats. With the judicious use of partitioning, very fast solid state drives, and more RAM than you can shake an actual ram at, indexes just aren't needed. And that's one less bloody thing to worry about! Phew!

On the other hand, the inability to codify a primary key may cause some angst, so the backend ETL process should take this into account prior to loading data into the database. Naturally, how much you care about this depends on the type of data you're loading into the database as well as the SQL queries used to wrench the data from the underlying table(s). Please have a conversation with your sensational Hadoop Administrator to determine the best course of action.

`CREATE TABLE` Statement (`KUDU`)

Any table created using the `KUDU` storage format automatically allows for `DELETE`s, `UPDATE`s and `UPSERT`s. We discuss `DELETE`s, `UPDATE`s and `UPSERT`s in the section below.

```
CREATE TABLE database_name.table_name
 (
  column_name_1 data_type_1 kudu_column_attr_1 COMMENT 'column-comment-1',
  column_name_2 data_type_2 kudu_column_attr_2 COMMENT 'column-comment-2',
  ...
  column_name_n data_type_n kudu_column_attr_n COMMENT 'column-comment-n'
  PRIMARY KEY (column_name_i,...,column_name_k)
 )
 PARTITION BY kudu_partition_clause
 COMMENT 'table-comment'
 STORED AS KUDU
 TBLPROPERTIES (
                'key-1','value-1',
                'key-2','value-2',
                ...
                'key-r','value-r'
               )
;
```

Although we discuss partitioning related to the `TEXTFILE` and `PARQUET` storage formats in *Chapter 16 – SQL Performance Improvements*, please note that the keyword is `PARTITION BY` for the `KUDU` storage format, but `PARTITIONED BY` for the `TEXTFILE` and `PARQUET` storage formats.

Although not technically necessary, you're probably going to use the `DELETE`, `UPDATE` and `UPSERT` commands with your `KUDU` table to mimic the functionality in your legacy database, so you may want to include a `PARTITION BY` Clause to break up the data so Hadoop can parallelize your queries.

At its very simplest, you can create a table stored using the `KUDU` storage format by specifying a primary key with the `PRIMARY KEY` Clause as well as the partitioning scheme using the `PARTITION BY` Clause. Below, we are specifying that the `POSTAL_CODE` column is the primary key and that we'd like the table slashed up into `4` parts using the `HASH` partitioning method.

```
USE PROD_SCHEMA;
CREATE TABLE DIM_POSTAL_CODE_PART(
                                  POSTAL_CODE STRING,
                                  CITY        STRING,
                                  STATE_CODE  STRING,
                                  LATITUDE    DOUBLE,
                                  LONGITUDE   DOUBLE,<-------- WHOA!! COMMA!!
                                  PRIMARY KEY (POSTAL_CODE)
                                 )
 PARTITION BY HASH (POSTAL_CODE) PARTITIONS 4
 STORED AS KUDU;
```

At this point, you're table is ready to insert into:

```
INSERT INTO DIM_POSTAL_CODE_PART
 SELECT *
  FROM DIM_POSTAL_CODE;
```

Note that if you describe the table using the formatted option, you'll notice that there's a new *serde*, *input format* and *output format* which the `STORED AS KUDU` Clause sets for us:

```
| SerDe Library:          | org.apache.hadoop.hive.kudu.KuduSerDe          | NULL          |
| InputFormat:            | org.apache.hadoop.hive.kudu.KuduInputFormat    | NULL          |
| OutputFormat:           | org.apache.hadoop.hive.kudu.KuduOutputFormat   | NULL          |
```

We talk more about partitioning in the *Chapter 16 – SQL Performance Improvements*.

Now that you have a `KUDU` table ready to use, you can perform `DELETE`s, `UPDATE`s and `UPSERT`s on it. The next section describes this in more detail.

DELETE,UPDATE and UPSERT (KUDU ONLY)

Since the `KUDU` storage format allows for deletes and updates, the DML commands `DELETE`, `UPDATE` and `UPSERT` apply only to it and not to tables created with the `TEXTFILE` or `PARQUET` storage formats. (This is a slight lie since it depends on the version of Hadoop you are running. More on this later.)

If you want to delete all of the rows from a `KUDU` table, effectively truncating the table, you can use the `DELETE` Statement without a `WHERE` Clause:

```
DELETE
 FROM PROD_SCHEMA.DIM_POSTAL_CODE;
```

Since there's no corresponding `COMMIT` Statement in Hadoop, the rows are deleted with abandon.

You can limit the rows deleted by adding a `WHERE` Clause to the `DELETE` Statement. For example, let's delete all of the rows where the first three numbers of the postal code equal `999`:

```
DELETE
 FROM PROD_SCHEMA.DIM_POSTAL_CODE
 WHERE SUBSTR(POSTAL_CODE,1,3)='999';
```

The general syntax for this form of the `DELETE` Statement is as follows:

```
DELETE
 FROM database_name.table_name
 WHERE where-condition;
```

There's a second form of the `DELETE` Statement which allows you to join to another table (`KUDU`, `TEXTFILE` or `PARQUET`). The general syntax for this form of the `DELETE` Statement is as follows:

```
DELETE database_name.table_name_1
 FROM database_name.table_name_1 JOIN database_name.table_name_2
 ON join-criteria
 WHERE where-condition;
```

Note that *`table_name_1`* must be a `KUDU` table, but *`table_name_2`* can be `KUDU`, `TEXTFILE` or `PARQUET`.

For example, let's remove rows from `DIM_POSTAL_CODE` using a second table containing keys used for the deletion. In this example, the `WHERE` Clause is not necessary:

```
DELETE PROD_SCHEMA.DIM_POSTAL_CODE
 FROM PROD_SCHEMA.DIM_POSTAL_CODE A JOIN PROD_SCHEMA.BAD_POSTAL_CODES B
 ON A.POSTAL_CODE=B.POSTAL_CODE;
```

Next, let's assume there's an additional `STRING` column in the table `BAD_POSTAL_CODES` named `OK_TO_DELETE` which is set to `Y` if it's okay to delete those particular postal codes from `DIM_POSTAL_CODE`. In this case, a `WHERE` Clause is necessary:

```
DELETE PROD_SCHEMA.DIM_POSTAL_CODE
 FROM PROD_SCHEMA.DIM_POSTAL_CODE A JOIN PROD_SCHEMA.BAD_POSTAL_CODES B
 ON A.POSTAL_CODE=B.POSTAL_CODE
 WHERE B.OK_TO_DELETE='Y';
```

Now, deleting data can be a depressing thing, so let's talk about updates and more happy times. You can update rows of a `KUDU` table simply by providing the columns you want to update. For example, to update a single column across the entire table, you can code this:

```
UPDATE PROD_SCHEMA.BAD_POSTAL_CODES
 SET OK_TO_DELETE='Y';
```

Alternatively, you can limit the extent of the updates by providing a `WHERE` Clause:

```
UPDATE PROD_SCHEMA.BAD_POSTAL_CODES
 SET OK_TO_DELETE='Y'
 WHERE UNATTRACTIVE_CITIZENS='Y';
```

In general, the syntax for this form of the `UPDATE` Statement is as follows:

```
UPDATE database_name.table_name
 SET column-1 = value-1,
     column-2 = value-2,
     ...
     column-n = value-n
 WHERE where-condition;
```

Similar to the second form of the `DELETE` Statement, the `UPDATE` Statement allows you to join to another table (`KUDU`, `TEXTFILE` or `PARQUET`). The syntax for this form of the `UPDATE` Statement is as follows:

```
UPDATE database_name.table_name_1
 SET column-1 = value-1,
     column-2 = value-2,
     ...
     column-n = value-n
 FROM database_name.table_name_1 JOIN database_name.table_name_2
 ON join-criteria
 WHERE where-condition;
```

Let's assume that the table `BAD_POSTAL_CODES` contains a corrected two-letter state code column named `CORRECTED_TWO_LETTER_STATE_CODE` and let's fix our screwed up `DIM_POSTAL_CODE` table:

```
UPDATE PROD_SCHEMA.DIM_POSTAL_CODE
 SET A.STATE_CODE=B.CORRECTED_TWO_LETTER_STATE_CODE
 FROM PROD_SCHEMA.DIM_POSTAL_CODE A JOIN PROD_SCHEMA.BAD_POSTAL_CODES B
 ON A.POSTAL_CODE=B.POSTAL_CODE
 WHERE B.UNATTRACTIVE_CITIZENS IN ('Y','N');
```

Occasionally, you'll create, or be given, a table containing a combination of data to insert into a table as well as data that can be used to update the same table. Effectively, you could make two passes through this fab table, once using an `INSERT` Statement and again using an `UPDATE` Statement. But, what fun would that be when you can

take care of both simultaneously using the `UPSERT` Statement! The `UPSERT` Statement makes use of the primary key of the `KUDU` table and will add additional rows to the table where the primary key is new, but update the non-primary key columns where the primary key exists. For example, let's use the `UPSERT` Statement on the `DIM_POSTAL_CODE` table:

```
UPSERT INTO PROD_SCHEMA.DIM_POSTAL_CODE
 SELECT *
  FROM DIM_POSTAL_CODE_NEW_AND_FIXES;
```

The general syntax for the `UPSERT` Statement is as follows:

```
UPSERT INTO database_name.table_name_1(column-1,
                                       column-2,
                                       ...,
                                       column-n)
 SELECT *
  FROM database_name.table_name_2
  WHERE where-condition;
```

If the mood strikes you, you can also use the `VALUES` Clause with the `UPSERT` Statement:

```
UPSERT INTO database_name.table_name_1(column-1,
                                       column-2,
                                       ...,
                                       column-n)
 VALUES(value-1,value-2,...,value-n);
```

DELETE, UPDATE and UPSERT (NON-KUDU TABLES)

Recall I mentioned earlier that you can delete and update `PARQUET` and `TEXTFILE` tables. This requires you to use the `ORC` storage format. We avoid discussion of this storage format in the book. Please see your Hadoop documentation on how to delete and update non-`KUDU` tables.

ALTER TABLE Statement (TEXTFILE/PARQUET/KUDU)

Once your table has been created, it would be a damned shame to have to re-create it just to add a new column, or change a column's name, or just rename the table itself. Well, my friends, you can sleep well at night because the `ALTER TABLE` Statement allows you to do all of these things and more! In this section, we avoid talking about partition changes until *Chapter 16 – SQL Performance Improvements*.

To rename a table, you can use the following syntax:

```
ALTER TABLE original-table-name RENAME TO new-table-name;
```

For example, let's rename the table `DIM_POSTAL_CODE2` to `DIM_POSTAL_CODE_BACKUP_1`:

```
[hdpserver.com:21000] prod_schema> ALTER TABLE DIM_POSTAL_CODE2 RENAME TO
                                                    DIM_POSTAL_CODE_BACKUP_1;
Query: ALTER TABLE DIM_POSTAL_CODE2 RENAME TO DIM_POSTAL_CODE_BACKUP_1
+--------------------------+
| summary                  |
+--------------------------+
| Renaming was successful. |
+--------------------------+
```

Now, when you change the name of a managed table, the associated directory in HDFS is also altered to reflect the change:

```
[hdpserver.com:21000] prod_schema> DESC FORMATTED DIM_POSTAL_CODE_BACKUP_1;
...snip...
Location: hdfs://lnxserver.com:8020/warehouse/tablespace/
                                         managed/hive/dim_postal_code_backup_1

Table Type: MANAGED_TABLE
...snip...
```

But, this is not true for external tables. The table's name will be changed, but the underlying directory remains the same. For example, let's create an external table called `EXT_POSTAL_CODE`:

```
CREATE EXTERNAL TABLE EXT_POSTAL_CODE STORED AS PARQUET AS
 SELECT *
  FROM DIM_POSTAL_CODE;
```

The `Location` for this external table is:

```
Location: hdfs://lnxserver.com:8020/warehouse/tablespace/
                                                 external/hive/EXT_POSTAL_CODE
```

Now, let's rename this table to `TMP_POSTAL_CODE`:

```
[hdpserver.com:21000] prod_schema> ALTER TABLE EXT_POSTAL_CODE RENAME TO
                                                              TMP_POSTAL_CODE;
Query: ALTER TABLE EXT_POSTAL_CODE RENAME TO TMP_POSTAL_CODE
+--------------------------+
| summary                  |
+--------------------------+
| Renaming was successful. |
+--------------------------+
```

And, the `Location` for the external table `TMP_POSTAL_CODE` is still the original HDFS location:

```
Location: hdfs://lnxserver.com:8020/warehouse/tablespace/
                                                 external/hive/EXT_POSTAL_CODE
```

Recall that the `DIM_POSTAL_CODE` table contains the following columns:

```
+-------------+--------+---------------------+
| name        | type   | comment             |
+-------------+--------+---------------------+
| postal_code | string | 5-DIGIT POSTAL CODE |
| city        | string | CITY NAME           |
| state_code  | string | 2-LETTER STATE CODE |
| latitude    | double | LATITUDE            |
| longitude   | double | LONGITUDE           |
+-------------+--------+---------------------+
```

Since both the `LATITUDE` and `LONGITUDE` columns are computed at the centroid of each `POSTAL_CODE`, let's rename these two columns to reflect that important information:

```
[hdpserver.com:21000] prod_schema> ALTER TABLE DIM_POSTAL_CODE_BACKUP_2 CHANGE
                                        LONGITUDE LONGITUDE_CENTROID DOUBLE;
Query: ALTER TABLE DIM_POSTAL_CODE_BACKUP_2 CHANGE LONGITUDE LONGITUDE_CENTROID
DOUBLE
+--------------------------+
```

```
| summary                  |
+--------------------------+
| Column has been altered. |
+--------------------------+

[hdpserver.com:21000] prod_schema> ALTER TABLE DIM_POSTAL_CODE_BACKUP_2 CHANGE
                                   LATITUDE LATITUDE_CENTROID DOUBLE;
Query: ALTER TABLE DIM_POSTAL_CODE_BACKUP_2 CHANGE LATITUDE LATITUDE_CENTROID
DOUBLE
+--------------------------+
| summary                  |
+--------------------------+
| Column has been altered. |
+--------------------------+
```

And, let's describe the table again:

```
+--------------------+--------+---------+
| name               | type   | comment |
+--------------------+--------+---------+
| postal_code        | string |         |
| city               | string |         |
| state_code         | string |         |
| latitude_centroid  | double |         |
| longitude_centroid | double |         |
+--------------------+--------+---------+
```

In general, the syntax for renaming a column is as follows:

```
ALTER TABLE table-name CHANGE column-name new_column-name column-data-type;
```

Not only can you change the column's name, but its data type as well. In the examples above, I specified the original data type of `DOUBLE`.

If you'd like to completely remove a column from a table, you can use `DROP COLUMN` syntax. In general,

```
ALTER TABLE table-name DROP COLUMN column-name;
```

For example, let's drop the `STATE_CODE` column the table `DIM_POSTAL_CODE_BACKUP_2`:

```
[hdpserver.com:21000] prod_schema> ALTER TABLE DIM_POSTAL_CODE_BACKUP_2
                                               DROP COLUMN STATE_CODE;
Query: ALTER TABLE DIM_POSTAL_CODE_BACKUP_2 DROP COLUMN STATE_CODE
+--------------------------+
| summary                  |
+--------------------------+
| Column has been dropped. |
+--------------------------+
```

And, after describing the table, you'll notice that the `STATE_CODE` column has been completely vaporized:

```
+--------------------+--------+---------+
| name               | type   | comment |
+--------------------+--------+---------+
| postal_code        | string |         |
| city               | string |         |
| latitude_centroid  | double |         |
| longitude_centroid | double |         |
+--------------------+--------+---------+
```

If you'd like to add one or more columns to an existing table, you can use the ADD COLUMN Syntax. In general, the syntax is as follows:

```
ALTER TABLE table-name ADD COLUMNS (column-name-1 data-type-1,
                                    column-name-2 data-type-2,
                                    ...,
                                    column-name-n data-type-n);
```

For example, let's add the column CITY_NAME as well as the column POP_IN_POSTAL_CODE:

```
[hdpserver.com:21000] prod_schema> ALTER TABLE DIM_POSTAL_CODE_BACKUP_2
                                       ADD COLUMNS (CITY_NAME STRING,
                                                    POP_IN_POSTAL_CODE BIGINT);
Query: ALTER TABLE DIM_POSTAL_CODE_BACKUP_2 ADD COLUMNS (CITY_NAME
STRING,POP_IN_POSTAL_CODE BIGINT)
+---------------------------------------------+
| summary                                     |
+---------------------------------------------+
| New column(s) have been added to the table. |
+---------------------------------------------+
```

Describing this table now yields:

```
+--------------------+--------+---------+
| name               | type   | comment |
+--------------------+--------+---------+
| postal_code        | string |         |
| city               | string |         |
| latitude_centroid  | double |         |
| longitude_centroid | double |         |
| city_name          | string |         |
| pop_in_postal_code | bigint |         |
+--------------------+--------+---------+
```

If you'd like to alter a table's or column's comment, you can use the COMMENT Statement. For tables, the syntax is:

```
COMMENT ON TABLE table-name IS 'table-comment';
```

And, for columns, the syntax is:

```
COMMENT ON COLUMN table-name.column-name IS 'column-comment';
```

Note that the maximum comment length, for either a table or a column, is 256 characters. For example, let's do up the table DIM_POSTAL_CODE_BACKUP_2 a treat:

```
[hdpserver.com:21000] prod_schema> COMMENT ON TABLE DIM_POSTAL_CODE_BACKUP_2
                                                   IS 'BACKUP TABLE #2';
+----------------+
| summary        |
+----------------+
| Updated table. |
+----------------+

[hdpserver.com:21000] prod_schema> COMMENT ON COLUMN
            DIM_POSTAL_CODE_BACKUP_2.POP_IN_POSTAL_CODE IS 'POPULATION (000s)';
+--------------------------+
| summary                  |
+--------------------------+
| Column has been altered. |
```

```
+---------------------------+
```

And, when describing the table using the formatted option, we see the following (some rows removed for clarity):

```
+------------------------------+------------+--------------------+
| name                         | type       | comment            |
+------------------------------+------------+--------------------+
| # col_name                   | data_type  | comment            |
|                              | NULL       | NULL               |
| postal_code                  | string     | NULL               |
| city                         | string     | NULL               |
| latitude_centroid            | double     | NULL               |
| longitude_centroid           | double     | NULL               |
| city_name                    | string     | NULL               |
| pop_in_postal_code           | bigint     | POPULATION (000s)  |
|                              | comment    | BACKUP TABLE #2    |
+------------------------------+------------+--------------------+
```

Note that the table's comment is placed to the right of the word `comment` under the `comment` column (how convenient!).

`CREATE VIEW` Statement

A *view* associates SQL syntax with a view name, similar to a table name, but without any underlying files in HDFS being created, similar to naming a child before it's conceived. A view can be used in the same locations a table name would go, such as after the keyword `FROM`. A view can be used to hide certain columns (e.g., social security number, creditcard number, etc.) from the users of the view as well as make writing SQL code more tidy. It's highly recommended that you create views from SQL code that's often used by you and your team, for consistency sake.

The simplified syntax for a view is as follows:

```
CREATE VIEW IF NOT EXISTS view-name AS
 SQL-select-query;
```

For example, let's create a view named `V_POSTAL_CODE_INFO` which joins the tables `DIM_POSTAL_CODE` and `DIM_US_STATE_MAPPING`:

```
CREATE VIEW PROD_SCHEMA.V_POSTAL_CODE_INFO AS
 SELECT A.POSTAL_CODE,A.CITY,A.LATITUDE,A.LONGITUDE,A.STATE_CODE,B.STATE_NAME
  FROM PROD_SCHEMA.DIM_POSTAL_CODE A LEFT JOIN
                                             PROD_SCHEMA.DIM_US_STATE_MAPPING B
  ON A.STATE_CODE=B.STATE_CODE
  WHERE A.STATE_CODE NOT IN ('AE','AP','AS','FM','GU','MH','MP','PW','VI');
```

You can describe the view just as if it were a table:

```
[hdpserver.com:21000] prod_schema > DESC V_POSTAL_CODE_INFO;
+-------------+--------+---------+
| name        | type   | comment |
+-------------+--------+---------+
| postal_code | string |         |
| city        | string |         |
| latitude    | double |         |
| longitude   | double |         |
| state_code  | string |         |
| state_name  | string |         |
+-------------+--------+---------+
```

Although the view shown above is very simple, views can be very complex including GROUP BY, HAVING and other clauses as well as CUBE, ROLLUP, analytic functions, etc. Now, if a view's SQL contains a WHERE Clause, as in the code shown above, when that view is used with a supplementary WHERE Clause, the two clauses are combined. In other words, the WHERE Clause as defined in the view's SQL code is **not** removed. For example, let's use the view V_POSTAL_CODE_INFO to pull in only Guam's postal codes. Note that the SQL code for the view itself excludes Guam data:

```
[hdpserver.com:21000] prod_schema> SELECT *
                                 >  FROM V_POSTAL_CODE_INFO
                                 >  WHERE STATE_CODE='GU';
Fetched 0 row(s) in 0.13s
```

As you see above, no rows are returned when using the view with a supplementary WHERE Clause indicating that the view's own WHERE Clause is still honored.

Now, when you've sickened with a view, you can just drop it:

```
DROP VIEW IF EXISTS view-name;
```

Using the SHOW and SET Statements

ImpalaSQL has two additional statements, SHOW and SET, which can be used to display important pieces of information, such as tables in the current database/schema, as well as set specific options in your ImpalaSQL session, such as the compression option.

To display the available databases, use SHOW DATABASES:

```
[hdpserver.com:21000] prod_schema> SHOW DATABASES;
+------------------+----------------------------------------------+
| name             | comment                                      |
+------------------+----------------------------------------------+
| _impala_builtins | System database for Impala builtin functions |
| default          | Default Hive database                        |
| prod_schema      | Bob Smith`s Department database              |
+------------------+----------------------------------------------+
```

Note that, although you can see the list of databases, you may not have permission to access them. Please talk to your lovely Hadoop Administrator if you'd like access to one or more databases.

To display the list of available tables in the current database, use SHOW TABLES:

```
[hdpserver.com:21000] prod_schema> SHOW TABLES;
+---------------------------+
| name                      |
+---------------------------+
| bob1                      |
| bob2                      |
| bob3                      |
| bob4                      |
| candybar_consumption_data |
| dim_calendar              |
| dim_postal_code           |
...snip...
| state_code_jamboree       |
| state_code_jamboree2      |
| state_code_jamboree3      |
| tacobellinfo              |
```

```
| tmp_postal_code            |
| zzz1                       |
| zzz4                       |
| zzz5                       |
+----------------------------+
```

Now, both `SHOW DATABASES` and `SHOW TABLES` allow you to specify a regular expression *pattern* to subset the displayed list of objects. For example, let's use `SHOW TABLES` and limit to just the dimension tables:

```
[hdpserver.com:21000] prod_schema> SHOW TABLES LIKE 'dim*';
+---------------------------+
| name                      |
+---------------------------+
| dim_calendar              |
| dim_postal_code           |
| dim_postal_code_backup_1  |
| dim_postal_code_backup_2  |
| dim_postal_code_part      |
| dim_us_state_mapping      |
+---------------------------+
```

Take note that, since table names are stored in lowercase in the database metadata, you'll have to use lowercase for the pattern. Oh, don't accidentally mix up this `LIKE` Clause in SQL with the `LIKE` Clause for `SHOW TABLES`!! The `LIKE` Clause in SQL uses the percent sign (`%`) to indicate one or more characters whereas the `LIKE` Clause for `SHOW TABLES` uses the asterisk (`*`).

You may occasionally be interested in seeing the SQL used to create a table or view. You can use the `SHOW CREATE TABLE` Statement to display a table's `CREATE TABLE` syntax as well as a view's underlying SQL. For example, let's use `SHOW CREATE TABLE` on the table `DIM_CALENDAR` (output cleaned up slightly):

```
[hdpserver.com:21000] prod_schema> SHOW CREATE TABLE DIM_CALENDAR;
CREATE TABLE prod_schema.dim_calendar (
 date_id DATE,
 day TINYINT,
 month TINYINT,
 year INT,
 quarter TINYINT,
 yyyyddd STRING,
 ddd STRING,
 first_day_of_month DATE,
 first_day_of_quarter DATE,
 first_day_of_year DATE,
 month_name STRING,
 weekday_name STRING,
 yyyyqq STRING,
 yyyymm STRING,
 yyyymmdd STRING,
 date_long STRING,
 date_short STRING
)
STORED AS PARQUET
LOCATION 'hdfs://lnxserver.com:8020/warehouse/tablespace/managed/
                                                             hive/dim_calendar'
TBLPROPERTIES(
              'OBJCAPABILITIES'='HIVEMANAGEDINSERTREAD,HIVEMANAGEDINSERTWRITE',
              'STATS_GENERATED'='TASK',
              'impala.events.catalogServiceId'='---:---',
              'impala.events.catalogVersion'='41',
              'impala.lastComputeStatsTime'='1648475816',
```

```
                 'numRows'='31',
                 'totalSize'='5892',
                 'transactional'='true',
                 'transactional_properties'='insert_only'
                )
```

Let's do a similar thing for the view `V_POSTAL_CODE_INFO` (output cleaned up slightly):

```
[hdpserver.com:21000] prod_schema> SHOW CREATE TABLE V_POSTAL_CODE_INFO;
CREATE VIEW `prod_schema`.v_postal_code_info AS
 SELECT A.POSTAL_CODE,
        A.CITY,
        A.LATITUDE,
        A.LONGITUDE,
        A.STATE_CODE,
        B.STATE_NAME
  FROM `prod_schema`.dim_postal_code a
   LEFT OUTER JOIN `prod_schema`.dim_us_state_mapping b
   ON A.STATE_CODE = B.STATE_CODE
  WHERE A.STATE_CODE NOT IN (
                             'AE',
                             'AP',
                             'AS',
                             'FM',
                             'GU',
                             'MH',
                             'MP',
                             'PW',
                             'VI'
                            )
```

Although we talk about partitioning in *Chapter 16 – SQL Performance Improvements*, you can use `SHOW PARTITIONS` to display the list of associated partitions for a table:

```
[hdpserver.com:21000] prod_schema> SHOW PARTITIONS DIM_POSTAL_CODE_PART;
+------------+-------+--------+---------+----------------------------------------+
| state_code | #Rows | #Files | Size    | Location                               |
+------------+-------+--------+---------+----------------------------------------+
| AE         | -1    | 1      | 1.14KB  | .../dim_postal_code_part/state_code=AE |
| AK         | -1    | 1      | 9.96KB  | .../dim_postal_code_part/state_code=AK |
...snip...
| WV         | -1    | 1      | 29.84KB | .../dim_postal_code_part/state_code=WV |
| WY         | -1    | 1      | 7.61KB  | .../dim_postal_code_part/state_code=WY |
| Total      | -1    | 61     | 1.31MB  |                                        |
+------------+-------+--------+---------+----------------------------------------+
```

And you can display a list of the underlying files associated with a table by using `SHOW FILES`:

```
[hdpserver:21000] prod_schema> SHOW FILES IN PROD_SCHEMA.DIM_POSTAL_CODE;
+-------------------------------------------------------------------------------------+----------+
| Path                                                                                | Size     |
+-------------------------------------------------------------------------------------+----------+
| hdfs://hdpserver/data/prod/teams/prod_schema/dim_postal_code/a1117506544_data.0.parq | 132.33MB |
| hdfs://hdpserver/data/prod/teams/prod_schema/dim_postal_code/a1011983093_data.0.parq | 24.68MB  |
+-------------------------------------------------------------------------------------+----------+
```

Now, the `SET` Statement allows you to modify how your queries are processed. With the exception of the options listed below, I would talk to your brainy Hadoop Administrator before modifying the other options, especially those related to memory and file size. To see the current values of the options (current/default values are displayed in brackets), just submit the `SET` Statement alone:

```
[hdpserver.com:21000] prod_schema> SET;
Query options (defaults shown in []):
```

```
                ABORT_ON_ERROR: [0]
                COMPRESSION_CODEC: []
                DEFAULT_FILE_FORMAT: [PARQUET]
                ...snip...
                THREAD_RESERVATION_AGGREGATE_LIMIT: [0]
                THREAD_RESERVATION_LIMIT: [3000]
                TIMEZONE: [America/New_York]

        Advanced Query Options:
                APPX_COUNT_DISTINCT: [0]
                BROADCAST_BYTES_LIMIT: [34359738368]
                BUFFER_POOL_LIMIT: []
                ...snip...
                SHUFFLE_DISTINCT_EXPRS: [1]
                SUPPORT_START_OVER: [false]
                TOPN_BYTES_LIMIT: [536870912]
        Shell Options
                WRITE_DELIMITED: False
                VERBOSE: True
                LIVE_SUMMARY: False
                OUTPUT_FILE: None
                DELIMITER: \t
                LIVE_PROGRESS: True

        Variables:
                No variables defined.
```

Note that additional (unfinalized) development options can be seen by using the `SET ALL` Statement instead:

```
[hdpserver.com:21000] prod_schema> SET ALL;
...snip...
Development Query Options:
        ALLOW_ERASURE_CODED_FILES: [0]
        BATCH_SIZE: [0]
        CPU_LIMIT_S: [0]
        ...snip...
        PLANNER_TESTCASE_MODE: [0]
        SPOOL_QUERY_RESULTS: [0]
        STRICT_MODE: [0]
...snip...
```

To set an option to a specific value, use the following syntax:

```
SET option=value;
```

Note that the value can be a specific word (such as `snappy`) or a Boolean value (such as `true` or `1` and `false` or `0`). For example, let's specify the compression codec option to `snappy`:

```
set compression_codec=snappy;
```

And, let's set `SYNC_DDL` to `1` to ensure that each SQL statement submitted to the cluster fully completes before the next SQL statement is tackled:

```
set sync_ddl=1;
```

And, finally, let's disable codegen if your SQL query previously received an illegal instruction or other hardware-related error message:

```
set disable_codegen=true;
```

Chapter 9 – ImpalaSQL Functions Parade

Continuing our discussing of SQL, in this chapter we focus on some useful functions available within ImpalaSQL. Specifically, we discuss aggregate functions – those functions used to compute summary statistics such as a count or average computed *down* (that is, *across*) rows of data – as well as functions computed *within* each row such as trimming blanks, getting rid of those damn tabs, and so on. We discuss analytic functions in the next chapter.

Basic Aggregate Functions

Just like your legacy database, ImpalaSQL features the familiar aggregate function culprits: `COUNT`, `AVG`, `MIN`, `MAX` and `SUM`. By default, each function is computed across an entire column's data, but this can be altered by providing the keyword `DISTINCT` to, say, compute the number of distinct values in the column. For the most part, you'll use these functions with a `GROUP BY` Statement unless you intend to go across all rows returned by your SQL query.

- `COUNT(`*`column-name`*`)` – This aggregate function computes the count of the non-null values in *`column-name`*.
    - You can provide the keyword `DISTINCT` to compute a distinct number of values of *`column-name`*.
    - If you just want to know the total number of rows, you can use the syntax `COUNT(*)` without providing *`column-name`*. For example, to count the total number of zip codes within each state in the table `prod_schema.dim_postal_code`, you can use `COUNT(*)`, like this:

```
select state_code,count(*) as nbr_zips
 from prod_schema.dim_postal_code
 group by state_code;
```

    - The example above assumes that the table `prod_schema.dim_postal_code` contains a distinct list of zip codes for each state. If this is not the case, we can use the `DISTINCT` keyword to remedy that situation, like this:

```
select state_code,count(distinct postal_code) as nbr_zips
 from prod_schema.dim_postal_code
 group by state_code;
```

    - You can use the `CASE` Expression within an aggregate function, such as `COUNT`. For example, in the code below, we're counting the distinct number of occurrences of `Hollywood` and `Broadway` with the help of a `CASE` Expression:

```
select count(distinct case
                       when postal_code='90027' then 'Hollywood'
                       when postal_code='10018' then 'Broadway'
                       else                          null
                      end) as culture_count
 from prod_schema.dim_postal_code;
```

- `AVG(`*`column-name`*`)` – This aggregate function computes the average of the non-null values of *`column-name`*.
    - You can provide the keyword `DISTINCT` to compute the average of the distinct number of values of *`column-name`*.
- `MIN(`*`column-name`*`)` – This aggregate function computes the minimum of the non-null values of *`column-name`*.
    - You can provide the keyword `DISTINCT` to compute the minimum of the distinct number of values of *`column-name`*…not that you would…
- `MAX(`*`column-name`*`)` – This aggregate function computes the maximum of the non-null values of *`column-name`*.
    - You can provide the keyword `DISTINCT` to compute the maximum of the distinct number of values of *`column-name`*…not that you would…

- `SUM(`*`column-name`*`)` – This aggregate function computes the sum of the non-null values of *`column-name`*.
    - You can provide the keyword `DISTINCT` to compute the sum of the distinct number of values of *`column-name`*.

Approximate Aggregate Functions

ImpalaSQL offers two very fast approximation functions, one which computes the median and another which computes the number of distinct values. Like telling everyone you're *almost a millionaire*, both functions return an approximate – not the real honest-to-goodness – value.

- `APPX_MEDIAN(`*`column-name`*`)` – This aggregate function computes the median of the non-null values in *`column-name`*.
    - You can provide the keyword `DISTINCT` to compute the median on the distinct number of values of *`column-name`*.
    - This is an approximation to reality.
- `NDV(`*`column-name`*`)` – This aggregate function computes the **n**umber of **d**istinct **v**alues and is an approximation to `COUNT(DISTINCT column-name)`.
    - You can provide the keyword `DISTINCT` to compute the number of distinct values of *`column-name`*.
    - This is an approximation to reality.

Statistical Aggregate Functions

Although `AVG`, `COUNT`, `MIN`, `MAX` and `SUM` can be considered light-and-fluffy statistical functions, ImpalaSQL offers more meaty functions which compute the standard deviation and variance.

- `STDDEV_SAMP(`*`column-name`*`)` – This aggregate function computes the **sample** standard deviation of the non-null values in *`column-name`*.
    - You can provide the keyword `DISTINCT` to compute the sample standard deviation on the distinct number of values of *`column-name`*.
    - The function name `STDDEV` is an alias for `STDDEV_SAMP`. In the interest of clarity, please don't use it.
    - Note that zero is returned if only one row is provided as input.
- `STDDEV_POP(`*`column-name`*`)` – This aggregate function computes the **population** standard deviation of the non-null values in *`column-name`*.
    - You can provide the keyword `DISTINCT` to compute the population standard deviation on the distinct number of values of *`column-name`*.
    - Note that zero is returned if only one row is provided as input.
- `VARIANCE_SAMP(`*`column-name`*`)` – This aggregate function computes the **sample** variance of the non-null values in *`column-name`*.
    - You can provide the keyword `DISTINCT` to compute the sample variance on the distinct number of values of *`column-name`*.
    - The function names `VARIANCE` and `VAR_SAMP` are aliases for `VARIANCE_SAMP`. In the interest of clarity, please don't use them.
    - Note that zero is returned if only one row is provided as input.
- `VARIANCE_POP(`*`column-name`*`)` – This aggregate function computes the **population** variance of the non-null values in *`column-name`*.
    - You can provide the keyword `DISTINCT` to compute the population variance on the distinct number of values of *`column-name`*.
    - The function name `VAR_POP` is an alias for `VARIANCE_POP`. In the interest of clarity, please don't use it.
    - Note that zero is returned if only one row is provided as input.

`GROUP_CONCAT` Aggregate Function

This aggregate function produces a delimited text string based on the rows of data down a single column. Since this is an aggregate function, it can be used with the `GROUP BY` Clause to produce several delimited text strings based on the grouping column(s).

- ☐ `GROUP_CONCAT(`*`column-name`*`)` – When using this syntax, the delimiter defaults to a comma followed by a single space: `', '`.
- ☐ `GROUP_CONCAT(`*`column-name`*`,'`*`separator`*`')` – When using this syntax, you can provide any desired delimiter.
- ☐ You can provide the keyword `DISTINCT` to the left of *`column-name`* to produce a delimited string of distinct values of *`column-name`*.

For example, to produce a comma-delimited list of postal codes in New Jersey and Pennsylvania, you can use `GROUP_CONCAT` as shown below. Note that I'm replacing the default delimiter `', '` with `','`.

```
select state_code,group_concat(postal_code,',') as zip_in_state
 from prod_schema.dim_postal_code
 where state_code in ('NJ','PA')
 group by state_code;

+------------+-----------------------------+
| state_code | zip_in_state                |
+------------+-----------------------------+
| NJ         | 07024,07054,07088,...snip... |
| PA         | 15004,15009,15030,...snip... |
+------------+-----------------------------+
```

Unlike variants found in other databases, there's no `ORDER BY` option. If you want the delimited data to be sorted, provide a subquery along with an `ORDER BY` Clause:

```
with vwDAT as (
               select state_code,postal_code
                from prod_schema.dim_postal_code
                where state_code in ('NJ','PA')
                order by state_code,postal_code
                limit 10000000
              )
select state_code,group_concat(postal_code,',') as zip_in_state
 from vwDAT
 group by state_code;

+------------+-----------------------------+
| state_code | zip_in_state                |
+------------+-----------------------------+
| PA         | 15001,15003,15004,...snip... |
| NJ         | 07001,07002,07003,...snip... |
+------------+-----------------------------+
```

Take note that the `LIMIT` Clause may be required when using an `ORDER BY` Clause in a subquery. Please ensure the number of rows specified by the `LIMIT` Clause exceeds the number of rows emanating from the `WITH` Clause.

Within-Row Functions

As you can well imagine, there are many functions available in ImpalaSQL. We won't go through every single one because that would drive you insane…and we don't want that! Note that the date- and time-related functions are located in *Chapter 10 – Voyage of the Damned (Dates & Times – ImpalaSQL Edition)* and the regular expression-

related functions (such as `REGEXP_EXTRACT()`, `REGEXP_REPLACE()`, etc.) appear in *Chapter 11 – Regular Expressions*. We begin with a quick list of functions you're probably already familiar with.

Functions You're Probably Already Familiar With

In the grid below, an asterisk (*) indicates that the data type returned from the function is the same as the data type of the input value.

MATHEMATICAL FUNCTIONS		
Function	**Return Type**	**Description**
abs(*v*)	*	Returns the absolute value of the input.
acos(*v*)	DOUBLE	Returns the arccosine of the input.
asin(*v*)	DOUBLE	Returns the arcsine of the input.
atan(*v*)	DOUBLE	Returns the arctangent of the input.
atan2(*o*,*a*)	DOUBLE	Returns the arctangent of the opposite/adjacent. (The documentation mistakenly calls this atan.)
cos(*v*)	DOUBLE	Returns the cosine of the input.
sin(*v*)	DOUBLE	Returns the sine of the input.
tan(*v*)	DOUBLE	Returns the tangent of the input.
cot(*v*)	DOUBLE	Returns the cotangent of the input.
cosh(*v*)	DOUBLE	Returns the hyperbolic cosine of the input.
sinh(*v*)	DOUBLE	Returns the hyperbolic sine of the input.
tanh(*v*)	DOUBLE	Returns the hyperbolic tangent of the input.
exp(*v*)	DOUBLE	Returns e^v.
floor(*v*)	*	Returns the integer just below (or equal to) the input.
ceil(*v*)	*	Returns the integer just above (or equal to) the input.
e()	DOUBLE	Returns e^1.
pi()	DOUBLE	Returns cherry π.
ln(*v*)	DOUBLE	Returns the natural (base e) logarithm of the input.
log2(*v*)	DOUBLE	Returns the base 2 logarithm of the input.
log10(*v*)	DOUBLE	Returns the base 10 logarithm of the input.
mod(*a*,*n*)	*	Returns the remainder after dividing *a* by *n*.
pow(*v*,*p*)	DOUBLE	Returns the input raised to the power *p*.
random() **rand()**	DOUBLE	Returns a random number between 0 and 1. Produces the same sequence for different queries.
random(*seed*) **rand(*seed*)**	DOUBLE	Returns a random number between 0 and 1. If a constant seed is provided, produces the same sequence for different queries. If a non-constant seed is provided, produces different sequence.
round(*v*,*d*)	*	Rounds the input value to *d* decimal places. If second argument is excluded, rounds to integer.
sign(*v*)	INT	Returns -1, if v<0; 0, if v=0; +1, if v>0.
sqrt(*v*)	DOUBLE	Returns the square root of the input.

BITWISE FUNCTIONS		
Function	**Return Type**	**Description**
bitand(*n*,*m*)	*	Returns the bitwise AND of the arguments. Can use the ampersand (&) instead: n&m
bitor(*n*,*m*)	*	Returns the bitwise OR of the arguments. Can use the vertical bar (|) instead: n|m
bitnot(*v*)	*	Returns the complement of the input. Can use the tilde (~) instead: ~v
bitxor(*n*,*m*)	*	Returns the Exclusive OR of the arguments. Can use the caret (^) instead: n^m
shiftleft(*n*,*d*)	*	Returns n shifted left by *d* bits.
shiftright(*n*,*d*)	*	Returns n shifted right by *d* bits.
countset(*n*)	*	Returns the number of 1 bits in *n*. You may know this better as the population count or popcount.
countset(*n*,0)	*	Same as countset(n), but returns the number of 0 bits in *n*. Note that countset(n,1) is the same as countset(n).

CONVERSION FUNCTIONS		
Function	**Return Type**	**Description**
cast(*expr* as *type*)	*type*	Returns *expr* casted as the data type *type*.
typeof(*expr*)	STRING	Returns the data type of *expr* as a STRING.

CONDITIONAL FUNCTIONS		
Function	**Return Type**	**Description**
coalesce(*v1*,*v2*,*v3*,…)	*	Returns first non-NULL value in the argument list.
decode(*expr*,*match1*,*result1*, *match2*,*result2*, …,…, *default*)	*	Bog-standard decode() function returns the corresponding *result#* if *expr* matches *match#*; otherwise, the *default* value is returned.
if(*cond*,*true*,*false*)	*true*'s Type	Marsh-standard if() function returns *true* value if the *cond* is true; otherwise, *false* value.

ifnull(*v*,*r*) **isnull(*v*,*r*)** **nvl(*v*,*r*)**	*	Returns *r* if *v* is NULL. (Three for the price of one?)
nullif(*expr1*,*expr2*)	*	Returns NULL if *expr1*=*expr2*; otherwise, *expr1* is returned.
nvl2(*expr*,*r_notnull*,*r_null*)	*r_notnull*'s Type	Returns *r_notnull* if *expr* is not NULL; otherwise, *r_null*.

STRING FUNCTIONS		
Function	**Return Type**	**Description**
ascii(*v*)	INT	Returns the ASCII code for the first character of the text string *v*.
chr(*v*)	STRING	Returns the ASCII character corresponding to the integer argument.
btrim(*v*), **btrim(*v*,'*char-list*')**	STRING	Returns *v* with spaces removed from **b**oth the left and right ends (first form). Returns *v* with *char-list* removed from **b**oth the left and right ends (second form).
ltrim(*v*), **ltrim(*v*,'*char-list*')**	STRING	Returns *v* with spaces removed from the **l**eft end (first form). Returns *v* with *char-list* removed from the **l**eft end (second form).
rtrim(*v*), **rtrim(*v*,'*char-list*')**	STRING	Returns *v* with spaces removed from the **r**ight end (first form). Returns *v* with *char-list* removed from the **r**ight ends (second form).
trim(*v*)	STRING	Returns *v* with spaces removed from both the left and right ends.
lower(*v*) **lcase(*v*)**	STRING	Returns *v* converted to lowercase.
upper(*v*) **ucase(*v*)**	STRING	Returns *v* converted to UPPERCASE.
initcap(*v*)	STRING	Returns *V* With First Letter Of Each Word Capitalized. All other characters are lowercase.
concat(*v1*,*v2*,…)	STRING	Returns the concatenation of strings *v1*, *v2*, etc. Sadly, if any *v#* is NULL, returns NULL.
concat_ws(*sep*,*v1*,*v2*,…)	STRING	Returns the concatenation of strings *v1*, *v2*, etc. delimited by *sep*. Sadly, if any *v#* is NULL, returns NULL.
char_length(*v*) **character_length(*v*)** **length(*v*)**	INT	Returns the character length of the input value.
substr(*v*,*start*,*len*) **substring(*v*,*start*,*len*)**	STRING	Returns a substring of the input value *v* starting at *start* for a length of *len*. If *len* is left off, the portion of the string from *start* to the end of the string is returned.
left(*v*,*len*) **subleft(*v*,*len*)**	STRING	Returns the left-most portion of the input value *v* for a length of *len*.
right(*v*,*len*) **subright(*v*,*len*)**	STRING	Returns the right-most portion of the input value *v* for a length of *len*.
reverse(*v*)	STRING	.redro esrever ni *v* snruteR
lpad(*v*,*len*,*padding*)	STRING	Returns *v* padded to a length of *len* padded on the left with the *padding* characters.
rpad(*v*,*len*,*padding*)	STRING	Returns *v* padded to a length of *len* padded on the right with the *padding* characters.
instr(*v*,*ss*) **instr(*v*,*ss*,*sp*)** **instr(*v*,*ss*,*sp*,*occ*)**	INT	Returns the position of the search string *ss* within the input value *v* starting from the first character (first form). If you provide the optional starting position *sp*, then the search starts at *sp*. If you provide the optional occurrence value *occ*, the position of the *occ*th occurrence will be returned instead of the default first occurrence.
repeat(*v*,*t*) **space(*t*)**	STRING	Returns the text in the input value *v* repeated *t* times. The function space(*t*) is equivalent to repeat(' ',*t*).
replace(*v*,*ss*,*rs*)	STRING	Returns the input value *v* with all occurrences of the search string *ss* replaced with the replacement string *rs*.
translate(*v*,*fc*,*rc*)	STRING	Returns the input value *v* with the individual occurrences in from-correspondence *fc* with the corresponding correspondences in the replacement-correspondence *rc*. If this is still confusing, send me your correspondence.

Next, we talk about the ImpalaSQL functions you're probably not familiar with.

Functions You're Probably Not Familiar With

In the grid below, an asterisk (*) indicates that the data type returned from the function is the same as the data type of the input value.

MATHEMATICAL FUNCTIONS		
Function	**Return Type**	**Description**
degrees(*v*)	DOUBLE	Converts *v* from radians to degrees. For example, degrees(pi()) returns 180.
radians(*v*)	DOUBLE	Converts *v* from degrees to radians. For example, radians(180) returns 3.141592653589793.
factorial(*v*)	BIGINT	Returns the factorial of *v*. You can also use the notation *v!* even if you're not shouting. The maximum value for *v* is 20 and anything larger returns an error. For example, factorial(4) returns 24 as does 4!, but factorial(21) returns ERROR: UDF ERROR: factorial 21! too large for BIGINT.
fmod(*v1*,*v2*)	*	Returns the remainder (similar to the mod function), but *v1* and *v2* can be FLOAT or DOUBLE. Be careful using this function since both FLOAT and DOUBLE are subject

		to rounding issues. For example, fmod(12.6,3) is computed initially as 12.6/3 which returns 4.2. But, decimal portion is removed leaving just the 4. Now, 4 times 3 is 12 leaving a remainder of 0.6, which is returned by fmod(12.6,3) function. Actually, that's a slight lie since fmod(12.6,3) returns 0.6000003814697266. Rounding's a bitch!
least(*v1*,*v2*,…)	*	Returns the smallest value from *v1*, *v2*, … For example, least(1.1,2.2,3.3) returns 1.1.
greatest(*v1*,*v2*,…)	*	Returns the largest value from *v1*, *v2*, … For example, greatest(1.1,2.2, 3.3) returns 3.3.
log(*b*,*v*)	DOUBLE	Returns the logarithm base *b* of the value *v*. For example, log(7,150) returns 2.574957171855434. Equivalent to ln(150)/ln(7). Take note of the order of the parameters in this function!
quotient(*v1*,*v2*)	BIGINT	Returns *v1*/*v2* with all fractional parts zapped into non-existence. For example, quotient(7.2,2.6) returns 3 (7~~.2~~/2~~.6~~=3~~.5~~=3). Similarly, quotient(8.2,2.6) returns 4 (8~~.2~~/2~~.6~~=4).
positive(*v*)	*	Returns *v* regardless if *v* is positive or negative.
negative(*v*)	*	Returns the sign of *v* flipped regardless of *v*'s feelings pro or con. For example, negative(-10) returns 10 and negative(10) returns -10.
is_nan(*v*)	BOOLEAN	Returns true if *v* is stored as not a number (NaN); otherwise, false. For example, is_nan(sqrt(-1)) returns true (pissing off the Complex Analysis fans out there).
is_inf(*v*)	BOOLEAN	Returns true if *v* is either stored as Infinity or -Infinity. For example, is_inf(1/0) returns true.
min_tinyint(),max_tinyint()	*	Returns the minimum/maximum value a tinyint data type can store.
min_smallint(),max_smallint()	*	Returns the minimum/maximum value a smallint data type can store.
min_int(),max_int()	*	Returns the minimum/maximum value an int data type can store.
min_bigint(),max_bigint()	*	Returns the minimum/maximum value a bigint data type can store.
precision(*v*)	INT	Indicates the number of digits needed to create a correctly sized DECIMAL(precision,scale) value from *v*.
scale(*v*)	INT	Indicates the number of digits to the right of the decimal point to create a correctly sized DECIMAL(precision,scale) value from *v*. For example, given the value 151.75, precision(151.75) returns 5 and scale(151.75) returns 2. This indicates that the value 151.75 can safely live its life as DECIMAL(5,2).
trunc(*v*,*d*) **dtrunc(*v*,*d*)** **truncate(*v*,*d*)**	*	Truncates the value *v* leaving *d* fractional digits. If *d* is left off, then all fractional digits are lopped off. For example, trunc(1.2345) returns 1, but trunc(1.2345,2) returns 1.23. The three functions listed are equivalent.

BITWISE FUNCTIONS		
Function	**Return Type**	**Description**
getbit(*v*,*p*)	INT	Returns either a 0 or 1 based on the desired position *p* (from the right) for the value *v* as a binary number. For example, getbit(16,4) returns a 1 since 16 is represented as **1**0000 in binary and the fourth **place** (from the right) is a 1. As you can probably guess, getbit(16,3) returns 0.
setbit(*v*,*p*,*b*)	*	Returns *v* with the bit in place *p* from the right set to *b* (0 or 1). For example, setbit(16,3,1) returns 24 since the binary value for 16 is 10000 and the third bit from the right is set to a 1: 1**1**000.
rotateright(*v*,*p*)	*	Returns *v* with its bits rotated right by *p* bits. Note that the bits on the right swing around back to the left side of the number rather than falling off as in shiftright. For example, rotateright(24,1) returns 12 because 24 is represented as 11000 and rotating right by one position returns 01100 or 12. Now, rotateright(1,1) returns the value -128.
rotateleft(*v*,*p*)	*	Similar to rotateright but in the opposite direction.

CONVERSION FUNCTIONS		
Function	**Return Type**	**Description**
cast(*expr* AS *type* FORMAT *pattern*)	type	Returns the string *expr* as the specified data type *type* (either a TIMESTAMP or DATE) by parsing *expr* using the format indicated in *pattern*. For example, cast('10-31-2022' AS timestamp FORMAT 'mm-dd-yyyy') returns a TIMESTAMP data type set to the value 2022-10-31 00:00:00. We talk more about this function as well as format patterns in *Chapter 10 – Voyage of the Damned (Dates & Times – ImpalaSQL Edition)*.
isfalse(*expr*)	BOOLEAN	Returns true if *expr* is false; otherwise, true. For example, isfalse(1=1) returns false. And, isfalse(1=2) returns true. Note that isfalse(null) returns false.
istrue(*expr*)	BOOLEAN	Returns true if *expr* is true; otherwise, false. For example, istrue(1=1) returns true. And, istrue(1=2) returns false. Note that istrue(null) returns false.
isnotfalse(*expr*)	BOOLEAN	Similar to istrue(*expr*) except isnotfalse(null) returns true.
isnottrue(*expr*)	BOOLEAN	Similar to isfalse(*expr*) except isnottrue(null) returns true.
nullvalue(*expr*)	BOOLEAN	Returns true if *expr* is null; otherwise, false.
nonnullvalue(*expr*)	BOOLEAN	Returns true if *expr* is not null; otherwise, false.
nullifzero(*expr*)	*	Returns null if *expr* evaluates to zero; otherwise, returns the result of the numeric *expr*.

STRING FUNCTIONS		
Function	**Return Type**	**Description**
base64encode(*v*)	STRING	Converts the string *v* to base 64 encoding. This is useful if you have a problematic string. For example, as lovely a country as the Côte d'Ivoire is, its name can be problematic due to that tiny spaceship over the first lowercase letter o. One way around that is to encode it using base64encode: base64encode("Côte d'Ivoire") returns Q8O0dGUgZCdJdm9pcmU= and no spaceship.
base64decode(*v*)	STRING	Decodes a base64 encoded value. For example, base64decode ('Q8O0dGUgZCdJdm9pcmU=') returns Côte d'Ivoire. Nice!
find_in_set(*v*,*l*)	INT	Returns the entry position of *v* within the comma-delimited list *l*. For example, find_in_set('betty','fred,wilma,barney,betty') returns 4 since betty is the fourth entry in the comma-delimited list. Searching for barney would return 3.
locate(*v1*,*v2*,*pos*)	INT	Returns the character position within *v2* of the search value *v1*. By default, *pos* is zero and the search starts from the first position in *v2*. For example, locate('barney', 'fred,wilma,barney,betty') returns 12 since barney is the twelfth character in the string 'fred,wilma,**b**arney,betty'. Similar to instr with *v1* and *v2* swapped.
parse_url(*url*,*part*)	STRING	Returns the portion of the URL *url* based on the *part* desired. Note that *part* can be one of the following (must be capitalized!): PROTOCOL, HOST, PATH, REF, AUTHORITY, FILE, USERINFO, or QUERY. For example, parse_url('https://www.pornhub.com/grannies_with_a_passion_for_squid',' PATH') returns /grannies_with_a_passion_for_squid.
parse_url(*url*,*part*,*key*)	STRING	Extending parse_url above, if you specify QUERY for *part*, you can retrieve the value based on the *key*. For example, parse_url('https://www.pornhub.com/grannies_with_a_passion_for_squid?acct=1234567890','QUERY','acct') returns 1234567890.
split_part(*v*,*del*,*ix*)	STRING	Based on the indicated delimiter *del*, split_part returns the *ix*th delimited piece from the left of the string *v*. For example, split_part('fred,**wilma**,barney,betty',',',**2**) returns wilma. Note that if *ix*th is negative, the search starts from the right: split_part('fred,wilma,**barney**,betty',',',**-2**) returns barney.
levenshtein(*str1*,*str2*) **le_dst(*str1*,*str2*)**	DOUBLE	Returns the Levenshtein edit distance between *str1* and *str2*. For example, let's change CAROL CHANNING into SNOOP DOGG using the Levenshtein function (and a high-powered magic wand): SELECT LEVENSHTEIN('CAROL CHANNING','SNOOP DOGG'); returns a value of 11 which is the minimum number of insertions, deletions and substitutions required to genetically alter CAROL CHANNING to SNOOP DOGG. Sadly, the voice still remains.
jaro_winkler_similarity (*str1*,*str2*) **jw_sim(*str1*,*str2*)**	DOUBLE	Returns the Jaro-Winkler similarity between *str1* and *str2* which is a value between 0 and 1, where 0 indicates no match and 1 indicates exact match. For example, let's compute the Jaro-Winkler similarity between the strings AETNA and AETNA, INC.: SELECT JARO_WINKLER_SIMILARITY('AETNA','AETNA, INC.'); returns a value of 0.8909 indicating the match is better than meh.
jaro_winkler_distance(*str1*,*str2*) **jw_dst(*str1*,*str2*)**	DOUBLE	Returns the Jaro-Winkler distance which is computed as 1 - jaro_winkler_similarity (*str1*,*str2*). In the example above, the returned value is 0.1091 indicating that the distance between the two text strings is meh close.
Before Jaro and Winkler met one cloudless, moonlit night on a wind-swept pier in Tuscany on that fateful day before the aliens attacked, Jaro had his own damn matching functions:		
jaro_distance(*str1*,*str2*) **jaro_dst(*str1*,*str2*)**	DOUBLE	Returns the Jaro Distance between *str1* and *str2*. For example, SELECT JARO_DISTANCE('AETNA','AETNA, INC.'); returns 0.1818.
jaro_similarity(*str1*,*str2*) **jaro_sim(*str1*,*str2*)**	DOUBLE	Returns the Jaro Similarity between *str1* and *str2*. For example, SELECT JARO_SIMILARITY('AETNA','AETNA, INC.'); returns 0.8182. As you see, the Jaro Similarity returns a slightly lower value than the Jaro-Winkler Similarity. This is because the Jaro-Winkler similarity function returns a larger value if the two strings have the same *leading* characters. This is controlled by an optional third parameter (which I didn't mention) to the Jaro-Winkler functions called the *scaling factor* which is set, by default, to 0.1. I've never had a reason to alter the scaling factor when using these functions.

ADDITIONAL FUNCTIONS		
Function	**Return Type**	**Description**
current_database()	STRING	Returns the name of the current database schema, such as prod_schema.
user()	STRING	Returns the name of the logged in user. For example, bobsmith or the user name given to the production account.
sleep(*ms*)	N/A	Sleeps for *ms* milliseconds per row returned by a query.
uuid()	STRING	Returns a universal unique identifier and looks suspiciously like the number on the back of a gift

		card. Each time you call `uuid()`, you get a completely different value. For example, `uuid()` returned `1188db8f-981f-4750-bcfc-20ab71fa1106` for me and is a $10 Starbucks gift card.

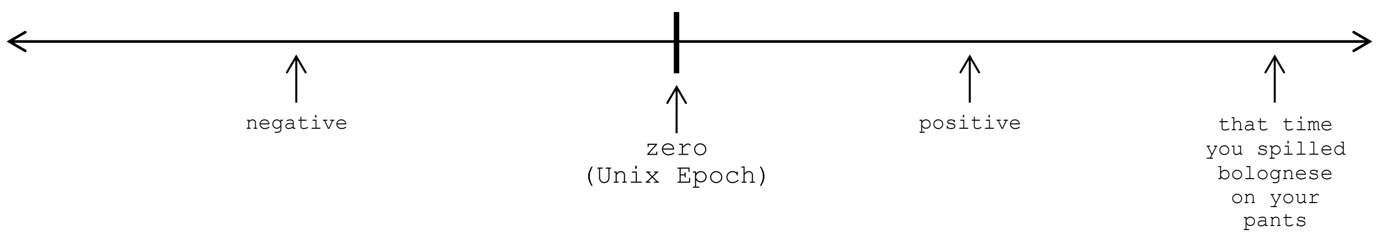
`version()`	STRING	Returns the Impala version information. For me, `version()` returns the following:
		`+------------------------------------------------------------------------------------------+`
		`| version()                                                                                |`
		`+------------------------------------------------------------------------------------------+`
		`| impalad version 3.4.0-SNAPSHOT RELEASE (build 27b919fc8a5907648349aa48eefc894e15a5a6d4) |`
		`| Built on Tue Aug  3 21:19:39 UTC 2021                                                    |`
		`+------------------------------------------------------------------------------------------+`

Chapter 10 – Voyage of the Damned (Dates & Times – ImpalaSQL Edition)

Yeah…sorry about this…but it has to be done…we need to talk about dates and times, the most frustrating, polyp-inducing topic in any database or programming language. Here goes nothin', dude!

When Nothing Means Something

As described previously, ImpalaSQL allows for both the `DATE` and `TIMESTAMP` data types. Both of these data types store their values internally as the number of days (for `DATE`) or seconds (for `TIMESTAMP`) since the pivot point `January 1, 1970`. This means that, internally, `January 1, 1970` equates to the big zot, the vast void of nothingness, another lonely Saturday night, or just plain ol' zero (`0`). Any date/timestamp subsequent to that is stored as a positive number, and any date/timestamp prior to that is negative. This pivot point is known as the *Unix epoch* and was chosen arbitrarily by the 1970s bell-bottom wearing creators of Unix.



DATE and TIMESTAMP Literals

You can quickly create a specific `DATE` or `TIMESTAMP` value by using literals. For example, to create a `DATE` data type using the DATE Literal, specify the keyword `DATE` followed by the desired date in `'yyyy-mm-dd'` format:

```
SELECT DATE '1962-03-21' AS TBDAY;

+------------+
| tbday      |
+------------+
| 1962-03-21 |
+------------+
```

You can use the DATE Literal syntax `DATE 'yyyy-mm-dd'` in any place where `DATE` data types are accepted.

Unfortunately, there's no TIMESTAMP Literal, but fear not, ImpalaSQL will convert a text string in the format `'yyyy-mm-dd hh:mi:ss.SSSSSS'` into a `TIMESTAMP` data type where `TIMESTAMP` data types are accepted. Note that you can leave off the slithering fractional seconds `.SSSSSS`. For example,

```
SELECT '1962-03-21 12:00:00' AS TBTIME;

+---------------------+
| tbtime              |
+---------------------+
| 1962-03-21 12:00:00 |
+---------------------+
```

Sadly, the value shown above is actually a text string, but will be evaluated as a `TIMESTAMP` when used with functions that expect a `TIMESTAMP` parameter. A similar automatic conversion occurs for text strings in `'yyyy-mm-dd'` format not using the DATE Literal `DATE 'yyyy-mm-dd'`.

Now, if you're like me, this discrepancy induces an uncontrollable facial t-t-t-tick. But, never fear, as you can use the `CAST` function to convert a text string to the desired `DATE` or `TIMESTAMP` data type and be done with it. Note

that you must still use the yyyy-mm-dd and yyyy-mm-dd hh:mi:ss.SSSSSS formats (although we revise that statement below):

```
SELECT CAST('1962-03-21' AS DATE) AS TBDAY;
+------------+
| tbday      |
+------------+
| 1962-03-21 |
+------------+

SELECT CAST('1962-03-21 12:00:00' AS TIMESTAMP) AS TBTIME;
+---------------------+
| tbtime              |
+---------------------+
| 1962-03-21 12:00:00 |
+---------------------+
```

The values displayed above are DATE and TIMESTAMP data types, respectively. You don't believe me, do you?

```
CREATE TABLE TACOBELLINFO STORED AS PARQUET AS
 SELECT CAST('1962-03-21' AS DATE) AS TBDAY,
        CAST('1962-03-21 12:00:00' AS TIMESTAMP) AS TBTIME

DESC TACOBELLINFO;
+--------+-----------+---------+
| name   | type      | comment |
+--------+-----------+---------+
| tbday  | date      |         |
| tbtime | timestamp |         |
+--------+-----------+---------+
```

Thhhpppttt! ☺

The CAST Function and Format Patterns

In the previous section, we learned that the DATE Literal requires the date string to be in yyyy-mm-dd format indicating a four-digit year followed by a dash followed by a two-digit month followed by a dash followed by a two-digit day all lovingly enveloped by quotes. We also learned that the CAST function can convert date and timestamp strings into DATE and TIMESTAMP data types assuming the strings are formatted properly. Recall the CAST function was described in *Chapter 9 – ImpalaSQL Functions Parade*. When working with dates and timestamps, the CAST function takes a FORMAT Clause followed by a *format pattern* allowing you to specify more complicated date and timestamp formats.

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
cast(*expr* as *type* format *pattern*)	DATE/ TIMESTAMP	Returns *expr* cast to the data type DATE or TIMESTAMP based on the format pattern specified in the FORMAT Clause.

Below are some of the format patterns you can use with the FORMAT Clause:

- ☐ yyyy – Four-digit year
- ☐ yy – Two-digit year. Since the two-digit century is missing, the 21st century is assumed, so 20yy is returned.
- ☐ rr – Two-digit year. If the year rr is between 00 and 49, the two century digits are 20. If rr is between 50 and 99, the two century digits are 19. For example, select cast('**00**-01-23' as date format 'rr-mm-dd'); returns **20**00-01-23 whereas select cast('**60**-01-23' as date format 'rr-mm-dd'); returns **19**60-01-23.
- ☐ mm – month number between 1 and 12.

- `MONTH`/`Month`/`month` – Full month name (`January`, `February`, etc.). Although the documentation indicates the correct case of these formats must be selected based on the incoming data, the case seems to be irrelevant and the correct conversion is performed. Please test this with your version of the software before proceeding.
- `MON/Mon/mon` – Three-letter month name (`Jan`, `Feb`, etc.). Although the documentation indicates the correct case of these formats must be selected based on the incoming data, the case seems to be irrelevant and the correct conversion is performed. Please test this with your version of the software before proceeding.
- `dd` – Two-digit day between `1` and `31`.
- `ddd` – Day of the year between `1` and `366`.
- `hh`/`hh12` – Hour of the day between `1` and `12` (12-hour clock).
- `hh24` – Hour of the day between `1` and `23` (24-hour/military clock).
- `mi` – Minute of the hour between `1` and `59`. **Don't confuse `mm` and `mi`!!**
- `ss` – Second of the minute between `1` and `59`.
- `am`/`a.m.`/`pm`/`p.m.` – The AM/PM indicators. It doesn't matter which one you use since they're synonymous.

For example, to convert the text `March 21, 1963` to a `DATE` data type, code this:

```
SELECT CAST('March 21, 1963' AS DATE
       FORMAT 'Month dd, yyyy') AS TBDAY;

+------------+
| tbday      |
+------------+
| 1963-03-21 |
+------------+
```

To convert the text `March 21, 1963 11:30 P.M.` to a `TIMESTAMP` data type, code this:

```
SELECT CAST('March 21, 1963 11:30 P.M.' AS TIMESTAMP
       FORMAT 'Month dd, yyyy hh:mi p.m.') AS TBTIME;

+---------------------+
| tbtime              |
+---------------------+
| 1963-03-21 23:30:00 |
+---------------------+
```

To convert `1963080`, made up of a four-digit year followed by a three-digit day of the year, to a `DATE` data type, code this:

```
SELECT CAST('1963080' AS DATE FORMAT 'yyyyddd') AS TBDAY;

+------------+
| tbday      |
+------------+
| 1963-03-21 |
+------------+
```

Unlike other databases, the day is not assumed to be the first of the month if the day of the month is not specified in the input:

```
SELECT CAST('1963-03' AS DATE FORMAT 'yyyy-mm') AS TBDAY;
ERROR: UDF ERROR: String to Date parse failed. Invalid string val: "1963-03"
```

One way around this is to concatenate `01` to your input text:

```
SELECT CAST(concat('1963-03','-01') AS DATE FORMAT 'yyyy-mm-dd') AS TBDAY;

+------------+
| tbday      |
+------------+
| 1963-03-01 |
+------------+
```

Note that the list of formats shown above can be considered as *input* formats; that is, formats used to convert from a string to a DATE/TIMESTAMP data type. There are additional formats which can be considered as *output* formats; that is, formats used to convert from DATE or TIMESTAMP data types to strings. For example, the format Q indicates the quarter of the year (1, 2, 3 or 4), but it can't be used with the examples shown above:

```
SELECT CAST('1963-1' AS DATE FORMAT 'yyyy-q') AS TBDAY;
ERROR: PARSE_ERROR: Quarter token is not allowed in a string to datetime conversion
```

So, the format Q cannot be used as an *input* format, but can be used as an *output* format. For example, the inner CAST function below converts 1963080 using the format yyyyddd to a DATE data type while the outer CAST function converts the DATE data type to a STRING data type in the format yyyy:q, shown below:

```
SELECT CAST(
            CAST('1963080' AS DATE FORMAT 'yyyyddd')<-- DATE data type
            AS STRING FORMAT 'yyyy:q'
           )
                                                     AS TBQTR;

+--------+
| tbqtr  |
+--------+
| 1963:1 |
+--------+
```

The additional *output* formats are as follows:

- q – Quarter of the year (1, 2, 3 or 4).
- ww – Week of the year (1 to 53).
- DAY/Day/day – full name of the day (Monday, Tuesday, ...). Note that the returned day is a full nine characters long including trailing spaces. Since WEDNESDAY is already nine characters long, it won't have any trailing spaces, but MONDAY will have three trailing spaces.
- DY/Dy/dy – abbreviated name of the day (Mon, Tue, ...). Unlike for the output format above, this format is exactly three characters long.

For example, let's create a **FRANKENDATE** using all four of the output formats shown above:

```
SELECT CAST(
            CAST('1963080' AS DATE FORMAT 'yyyyddd')<-- DATE data type
            AS STRING FORMAT 'yyyy:q/ww/DAY/Dy'
           )
                                                     AS TBFRANKENDATE

+-------------------------+
| tbfrankendate           |
+-------------------------+
| 1963:1/12/THURSDAY /Thu |
+-------------------------+
```

Unbelievable!!

In the output above, you'll notice that `THURSDAY` is followed by a single trailing blank exactly as expected for the `DAY` output format since it's forced to nine characters. To suppress blank padding, like that produced by the `DAY` format, place the `fm` format modifier directly before the `DAY` output format. For example, here's the output **without** the `fm` format modifier:

```
SELECT CAST(
            CAST('1963077' AS DATE FORMAT 'yyyyddd')
            AS STRING FORMAT 'yyyy:q/ww/DAY/Dy'
           )                                          AS TBFRANKENDATE;

+-------------------------+
| tbfrankendate           |
+-------------------------+
| 1963:1/11/MONDAY   /Mon |
+-------------------------+
```

And here is the output **with** the `fm` format modifier:

```
SELECT CAST(
            CAST('1963077' AS DATE FORMAT 'yyyyddd')
            AS STRING FORMAT 'yyyy:q/ww/fmDAY/Dy'
           )                                          AS TBFRANKENDATE;

+----------------------+
| tbfrankendate        |
+----------------------+
| 1963:1/11/MONDAY/Mon |
+----------------------+
```

Note that the case of some output formats makes a difference in the results. If you specify `DAY`, the name of the day is displayed in upper case (`MONDAY`). If you specify `Day`, proper case (`Monday`). And, `day`, lower case (`monday`). A similar comment holds for both `DY` and `MONTH`.

Finally, as you can guess from the examples above, characters such as a slash (/), dash (-) and the colon (:) are interpreted as themselves.

So, That's Format Patterns Done Then, Huh?

Unfortunately, the input and output format patterns described for `CAST()` in the previous section don't always work with other `DATE`/`TIMESTAMP`-related conversion functions. This makes me sad! ☹ Specifically, you need to be careful with the the following three functions:

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
`from_timestamp(`*`timestamp/string, output pattern`*`)`	`STRING`	Converts a `TIMESTAMP` to a `STRING` based on the provided output pattern. If the first argument is already a `TIMESTAMP`, the output pattern is applied to create the desired formatted string. If the first argument is a `STRING`, it must be in the appropriate `TIMESTAMP` format `yyyy-mm-dd hh:mi:ss.SSSSSS`. It will be converted internally to a `TIMESTAMP` and then the ouput pattern will be applied to create the desired formatted string.
`from_unixtime(`*`unixtime,output pattern`*`)`	`STRING`	The first argument contains a number of seconds since Unix Epoch and the second argument specified the desired output format.
`to_timestamp(`*`date_string,input pattern`*`)`	`TIMESTAMP`	Converts a *date_string* using the provided *input pattern* into a `TIMESTAMP`.

All three functions make use of the following format patterns (input or output):

- `yyyy` – Four-digit year
- `yy` – Two-digit year.
- `M` – month number between `1` and `12`, non-padded.
- `MM` – month number between `01` and `12`, zero-padded.
- `MMM` – month name between `Jan` and `Dec`.
- `d` – Day number, non-padded.
- `dd` – Day number, zero-padded.
- `H` – Hour between `1` and `12`, non-padded.
- `HH` – Hour between `01` and `12`, zero-padded.
- `m` – Minutes between `1` and `59`, non-padded.
- `mm` – Minutes between `01` and `59`, zero-padded.
- `s` – Seconds between `1` and `59`, non-padded.
- `ss` – Seconds between `01` and `59`, zero-padded.
- `S` – Fractional seconds.

For example, we can use the `TO_TIMESTAMP()` function to create a `TIMESTAMP` data type from a string:

```
SELECT TO_TIMESTAMP('Mar 21, 1963 12:00:00',
                    'MMM dd, yyyy HH:mm:ss') AS TS1;
+---------------------+
| ts1                 |
+---------------------+
| 1963-03-21 12:00:00 |
+---------------------+
```

Next, let's use the example above along with the `FROM_TIMESTAMP()` function and an output pattern to create an output string:

```
select FROM_TIMESTAMP(
                      TO_TIMESTAMP('Mar 21, 1963 12:00:00',
                                   'MMM dd, yyyy HH:mm:ss'),
                      'MMM dd, yyyy') AS STR1;
+--------------+
| str1         |
+--------------+
| Mar 21, 1963 |
+--------------+
```

Dealing with the Here and Now

The following functions return the current date as `DATE` or `TIMESTAMP`:

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
current_date()	DATE	Returns the current date as a `DATE` data type.
current_timestamp() **now()**	TIMESTAMP	Returns the current date and time as a `TIMESTAMP` data type.
timeofday()	STRING	Returns the current date and time as a `STRING`.

For example, the `CURRENT_DATE()` function returns the current date:

```
SELECT CURRENT_DATE();

+----------------+
| current_date() |
+----------------+
| 2022-02-12     |
+----------------+
```

The `CURRENT_TIMESTAMP()` and `NOW()` functions return the current date as well as time:

```
SELECT CURRENT_TIMESTAMP(),NOW();

+-------------------------------+-------------------------------+
| current_timestamp()           | now()                         |
+-------------------------------+-------------------------------+
| 2022-02-12 15:21:18.886155000 | 2022-02-12 15:21:18.886155000 |
+-------------------------------+-------------------------------+
```

Finally, the `TIMEOFDAY()` function returns a more elaborately formatted `STRING` including the timezone:

```
SELECT TIMEOFDAY();

+------------------------------+
| timeofday()                  |
+------------------------------+
| Sat Feb 12 15:22:35 2022 EST |
+------------------------------+
```

Hacking Out Pieces of DATEs/TIMESTAMPs Slasher Movie Stylie

There are several functions you can use to extract pieces of a `DATE` or `TIMESTAMP`, such as the month, hour, year and so on. Two functions, `EXTRACT()` and `DATE_PART()`, are generic and can extract several different pieces of information, but there are several functions (such as `YEAR()`) which pull out only one specific piece of information (such as the year as an `int`) from a `DATE` or `TIMESTAMP`.

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
extract(TIMESTAMP/DATE,*unit*) **extract(*unit* from TIMESTAMP/DATE)**	BIGINT	Returns the requested ***unit*** from a `TIMESTAMP` or `DATE` data type. You can use either form of the function.
date_part(*unit*,TIMESTAMP/DATE)	BIGINT	Returns the requested ***unit*** from a `TIMESTAMP` or `DATE` data type.

The ***unit*** indicated in the two functions above can be any of the following. Note that in the second form of `EXTRACT()`, ***unit*** does **not** need to be enclosed in quotes, but it does for the other forms.

- ☐ `EPOCH` – returns the Unix epoch
- ☐ `MILLISECOND` – returns the milliseconds portion of the `TIMESTAMP`
- ☐ `SECOND` – returns the seconds portion of the `TIMESTAMP`
- ☐ `MINUTE` – returns the minute portion of the `TIMESTAMP`
- ☐ `HOUR` – returns the hour portion of the `TIMESTAMP`
- ☐ `DAY` – returns the day portion of the `DATE` or `TIMESTAMP`
- ☐ `MONTH` – returns the month portion of the `DATE` or `TIMESTAMP`
- ☐ `QUARTER` – returns the quarter portion of the `DATE` or `TIMESTAMP`
- ☐ `YEAR` – returns the year portion of the `DATE` or `TIMESTAMP`

For example, given today's date, let's pull out the year and the quarter from it:

```
SELECT EXTRACT(YEAR FROM NOW()) AS YYYY,
       EXTRACT(QUARTER FROM NOW()) AS QUARTER

+------+---------+
| yyyy | quarter |
+------+---------+
| 2022 | 1       |
+------+---------+
```

Alternatively, you can use the following standalone functions to pull a specific portion from a DATE or TIMESTAMP:

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
day(TIMESTAMP/DATE)	INT	Returns the day number between 1 and 31.
dayofmonth(TIMESTAMP/DATE)	INT	Returns the day number between 1 and 31.
dayname(TIMESTAMP/DATE)	STRING	Returns the name of the day between Sunday to Saturday.
dayofweek(TIMESTAMP/DATE)	INT	Returns the day value and ranges between 1 (Sunday) and 7 (Saturday).
dayofyear(TIMESTAMP/DATE)	INT	Returns the day of the year value and ranges between 1 and 366.
hour(TIMESTAMP)	INT	Returns the hours value.
millisecond(TIMESTAMP)	INT	Returns the milliseconds value.
minute(TIMESTAMP)	INT	Returns the minutes value.
month(TIMESTAMP/DATE)	INT	Returns the month value.
monthname(TIMESTAMP/DATE)	STRING	Returns the month name between January and December.
quarter(TIMESTAMP/DATE)	INT	Returns the quarter value 1 (Jan/Feb/Mar), 2 (Apr/May/Jun), 3 (Jul/Aug/Sep) or 4 (Oct/Nov/Dec).
second(TIMESTAMP)	INT	Returns the seconds value.
week(TIMESTAMP/DATE)	INT	Returns the week of the year between 1 and 53.
weekofyear(TIMESTAMP/DATE)	INT	Returns the week of the year between 1 and 53.
year(TIMESTAMP/DATE)	INT	Returns the year value.

Again, let's pull out the year and quarter from today's date:

```
SELECT YEAR(NOW()) AS YYYY,
       QUARTER(NOW()) AS QUARTER

+------+---------+
| yyyy | quarter |
+------+---------+
| 2022 | 1       |
+------+---------+
```

Truncating DATEs and TIMESTAMPs

The functions TRUNC() and DATE_TRUNC() allow you to truncate a DATE or TIMESTAMP to a specific unit, such as month or year. Although these two functions are similar, the TRUNC() function allows for more units. Both functions return the same data type as the input. Note that for both functions, *unit* must be enclosed in quotes.

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
date_trunc(*unit*,TIMESTAMP/DATE)	TIMESTAMP/ DATE	Truncates the TIMESTAMP or DATE to the requested ***unit***. The ***unit*** can be one of the following: ▪ MICROSECONDS – Rounds a TIMESTAMP to the microsecond. ▪ MILLISECONDS – Rounds a TIMESTAMP to the millisecond. ▪ SECOND – Rounds a TIMESTAMP to the second. ▪ MINUTE – Rounds a TIMESTAMP to the minute. ▪ HOUR – Rounds a TIMESTAMP to the hour. ▪ DAY – Rounds a DATE/TIMESTAMP to the day. ▪ WEEK – Rounds a DATE/TIMESTAMP to the week. ▪ MONTH – Rounds a DATE/TIMESTAMP to the month. ▪ YEAR – Rounds a DATE/TIMESTAMP to the year. ▪ DECADE – Rounds a DATE/TIMESTAMP to the decade. ▪ CENTURY – Rounds a DATE/TIMESTAMP to the century. ▪ MILLENNIUM – Rounds a DATE/TIMESTAMP to the millennium.
trunc(TIMESTAMP/DATE,*unit*)	TIMESTAMP/ DATE	Truncates the TIMESTAMP or DATE to the requested ***unit***. The ***unit*** can be one of the following: ▪ SYYYY/YYYY/YEAR/SYEAR/ YYY/YY/Y – Rounds a DATE/TIMESTAMP to the year. ▪ Q – Rounds a DATE/TIMESTAMP to the quarter. ▪ MONTH/MON/MM/RM – Rounds a DATE/TIMESTAMP to the month. ▪ WW – Rounds a DATE/TIMESTAMP to the week. ▪ DDD/DD/J – Rounds a DATE/TIMESTAMP to the day. ▪ DAY/DY/D – Rounds a DATE/TIMESTAMP to the starting day of the week. ▪ HH/HH12/HH24 – Rounds a TIMESTAMP to the hour.

		▪ MI – Rounds a TIMESTAMP to the minute.

For example, let's say Taco Bell Industries was founded on March 21, 1963 11:30:35 A.M. (how appropriate…just before lunch…nom-nom!). Let's first turn that into a TIMESTAMP data type:

```
SELECT CAST('March 21, 1963 11:30:35 A.M.' AS TIMESTAMP
            FORMAT 'Month dd, yyyy HH:MI:SS AM') AS TB_FOUNDERS_DT;

+---------------------+
| tb_founders_dt      |
+---------------------+
| 1963-03-21 11:30:35 |
+---------------------+
```

Next, let's truncate to the beginning of the month:

```
SELECT TRUNC(
             CAST('March 21, 1963 11:30:35 A.M.' AS TIMESTAMP
                  FORMAT 'Month dd, yyyy HH:MI:SS AM'),
             'MONTH') AS TB_FOUNDERS_MNTH;

+---------------------+
| tb_founders_mnth    |
+---------------------+
| 1963-03-01 00:00:00 |
+---------------------+
```

As you see, the TRUNC() function, along with the ***unit*** set to MONTH, rounds the date and time to the first of the month.

Next, let's truncate to the beginning of the year:

```
SELECT TRUNC(
             CAST('March 21, 1963 11:30:35 A.M.' AS TIMESTAMP
                  FORMAT 'Month dd, yyyy HH:MI:SS AM'),
             'YEAR') AS TB_FOUNDERS_YEAR;

+---------------------+
| tb_founders_year    |
+---------------------+
| 1963-01-01 00:00:00 |
+---------------------+
```

And, as you could have probably predicted, the date and time have been truncated to the first day of the year.

Comparing DATEs and TIMESTAMPs

You can compare DATEs with DATEs and TIMESTAMPs with TIMESTAMPs to determine if they are identical or if one is ahead or behind the other.

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
DATE_CMP(DATE *dt1*,DATE *dt2*)	INT	Returns the following: $\begin{cases} -1, & if\ dt1 < dt2 \\ 0, & if\ dt1 = dt2 \\ +1, & if\ dt1 > dt2 \end{cases}$ NULL will be returned if either argument is NULL.

TIMESTAMP_CMP(TIMESTAMP *ts1*, TIMESTAMP *ts2*)	INT	Returns the following: $\begin{cases} -1, & if\ ts1 < ts2 \\ 0, & if\ ts1 = ts2 \\ +1, & if\ ts1 > ts2 \end{cases}$ NULL will be returned if either argument is NULL.

For example, let's test whether McDonald's Founder's Day (April 15, 1955) occurred on the same day as, prior to, or after Taco Bell's Founder's Day (March 21, 1963):

```
SELECT DATE_CMP(
                CAST('April 15, 1955' AS DATE
                     FORMAT 'Month, dd, yyyy'),
                CAST('March 21, 1963' AS DATE
                     FORMAT 'Month, dd, yyyy')
               ) AS AND_THE_WINNER_IS

+-------------------+
| and_the_winner_is |
+-------------------+
| -1                |
+-------------------+
```

Darn! It looks like McDonald's was founded prior to Taco Bell! Well, good things come to those who wait…like dysentery.

Computing Months/Days between DATEs and TIMESTAMPs

You can compute the number of days or months between two DATEs or TIMESTAMPs. Note that all three functions described below will return a negative value if the first argument occurs prior to the second argument.

Note: Just for shits-and-giggles, please take the time to check the return value from the ImpalaSQL calculation against a corresponding calculation executed on your legacy database during the course of your conversion.

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
months_between(TIMESTAMP/DATE, TIMESTAMP/DATE)	DOUBLE	Returns the number of full months between the two dates as a floating-point value. Note that the time portion is ignored in the calculation. The result will be negative if the first argument occurs prior to the second argument.
int_months_between(TIMESTAMP/DATE, TIMESTAMP/DATE)	INT	Similar to MONTHS_BETWEEN(), but returns the FLOOR() of the computed value. The result will be negative if the first argument occurs prior to the second argument.
datediff(TIMESTAMP/DATE,TIMESTAMP/DATE)	INT	Computes the number of days **between** the two arguments. Note that if you want the **total number of days** between the two dates, you must add one to the resulting value. The result will be negative if the first argument occurs prior to the second argument.

For example, let's see how many months there are between January 1, 2022 and February 1, 2022:

```
SELECT MONTHS_BETWEEN(DATE '2022-01-01',DATE '2022-02-01');

+--------------------------------------------------------+
| months_between(date '2022-01-01', date '2022-02-01') |
+--------------------------------------------------------+
| -1                                                     |
+--------------------------------------------------------+
```

The return value is negative because 2022-01-01 is prior to 2022-02-01. Let's try that again with the dates switched:

```
SELECT MONTHS_BETWEEN(DATE '2022-02-01',DATE '2022-01-01');

+----------------------------------------------------+
| months_between(date '2022-02-01', date '2022-01-01') |
+----------------------------------------------------+
| 1                                                  |
+----------------------------------------------------+
```

Now, what happens if we replace 2022-02-01 with 2022-02-02?

```
SELECT MONTHS_BETWEEN(DATE '2022-02-02',DATE '2022-01-01');

+----------------------------------------------------+
| months_between(date '2022-02-02', date '2022-01-01') |
+----------------------------------------------------+
| 1.032258064516129                                  |
+----------------------------------------------------+
```

Effectively, there's exactly one month plus one day between the two dates and it's that one additional day that's responsible for the fractional part. Since February 2022 has 28 days, you'd expect the fractional part to be equal to 1/28, but that's not true since MONTHS_BETWEEN() assumes each month to be 31 days long.

1/31 = 0.032258064516129

Let's do the same example above, but use INT_MONTHS_BETWEEN() instead:

```
SELECT INT_MONTHS_BETWEEN(DATE '2022-02-02',DATE '2022-01-01');

+--------------------------------------------------------+
| int_months_between(date '2022-02-02', date '2022-01-01') |
+--------------------------------------------------------+
| 1                                                      |
+--------------------------------------------------------+
```

And, as expected, the returned value is 1.

Next, let's use the DATEDIFF() function to compute the number of days between 2022-02-02 and 2022-01-01:

```
SELECT DATEDIFF(DATE '2022-02-02',DATE '2022-01-01');

+----------------------------------------------+
| datediff(date '2022-02-02', date '2022-01-01') |
+----------------------------------------------+
| 32                                           |
+----------------------------------------------+
```

The makes sense since we have all of January 2022 which accounts for 31 days plus the 2 days in February. Wait a minute!! That makes 33, but 32 is displayed above!! What gives, homie? The DATEDIFF() function computes the number of days **between two dates**, so you lose one day. For example, how many days are between today and tomorrow? Is it 1 or 2? That depends on what you're trying to calculate. Now, if you want to compute the **total number of days**, add one to the result:

```
SELECT DATEDIFF(DATE '2022-02-02',DATE '2022-01-01') + 1;

+------------------------------------------------+
| datediff(date '2022-02-02', date '2022-01-01') + 1 |
+------------------------------------------------+
| 33                                             |
+------------------------------------------------+
```

Finally, if you'd like to compute the number of seconds between two DATEs or TIMESTAMPs, you can make use of the UNIX_TIMESTAMP() function which, if you recall, returns the number of seconds since the Unix Epoch (January 1, 1970):

```
SELECT UNIX_TIMESTAMP('2022-02-02 00:00:00')
       - UNIX_TIMESTAMP('2022-01-01 00:00:00') AS SS ;

+---------+
| ss      |
+---------+
| 2764800 |
+---------+
```

Since there are 32 days between those two dates (as we've established above), the number of seconds can be checked like this:

$$32\ days\ \times 24 \frac{hours}{day} \times 60 \frac{minutes}{hour} \times 60 \frac{seconds}{minute} = 2{,}764{,}800\ seconds$$

Incredible!!

Shifting DATEs and TIMESTAMPs

In this section, we look into those functions which allow you to shift DATEs and TIMESTAMPs forward or backward by a certain amount, such as days or minutes. Although we talked about the DATE and TIMESTAMP data types earlier in the book, we held off talking about the INTERVAL keyword until now because several of the functions described in this section make use of it. Note that, unlike other databases, INTERVAL is not a data type and, therefore, cannot be used to define a column as such.

The INTERVAL keyword allows you to more easily specify a shift in DATEs or TIMESTAMPs using your *big boy* and *big girl* words rather than those filthy function things. An interval is specified using the following syntax:

INTERVAL ***amount units***

where ***amount*** indicates the number of ***units*** to shift by, and ***units*** is one of the following keywords:

- YEAR/YEARS – Shift a DATE/TIMESTAMP by years.
- MONTH/MONTHS – Shift a DATE/TIMESTAMP by months.
- WEEK/WEEKS – Shift a DATE/TIMESTAMP by weeks.
- DAY/DAYS – Shift a DATE/TIMESTAMP by days.
- HOUR/HOURS – Shift a TIMESTAMP by hours.
- MINUTE/MINUTES – Shift a TIMESTAMP by minutes.
- SECOND/SECONDS – Shift a TIMESTAMP by seconds.
- MILLISECOND/MILLISECONDS – Shift a TIMESTAMP by milliseconds.
- MICROSECOND/MICROSECONDS – Shift a TIMESTAMP by microseconds.
- NANOSECOND/NANOSECONDS – Shift a TIMESTAMP by nanoseconds.

Note that the plural forms above are there for those of us with obsessive compulsive disorder (OCD) who could never possibly code `INTERVAL 2 DAY` when we can code `INTERVAL 2 DAYS`. Phew!! OCD trigger averted!!

For example, let's shift today's date by `10` years in the future as well as the past:

```
SELECT CURRENT_DATE() AS TODAY,
       CURRENT_DATE() + INTERVAL 10 YEARS AS TEN_YEARS_IN_THE_FUTURE,
       CURRENT_DATE() - INTERVAL 10 YEARS AS TEN_YEAR_IN_THE_PAST

+------------+-------------------------+----------------------+
| today      | ten_years_in_the_future | ten_year_in_the_past |
+------------+-------------------------+----------------------+
| 2022-02-13 | 2032-02-13              | 2012-02-13           |
+------------+-------------------------+----------------------+
```

Note that the `INTERVAL` keyword also works with `TIMESTAMP`s as well:

```
SELECT NOW() AS TODAY,
       NOW() + INTERVAL 10 YEARS AS TEN_YEARS_IN_THE_FUTURE,
       NOW() - INTERVAL 10 YEARS AS TEN_YEAR_IN_THE_PAST

+-------------------------------+-------------------------------+-------------------------------+
| today                         | ten_years_in_the_future       | ten_year_in_the_past          |
+-------------------------------+-------------------------------+-------------------------------+
| 2022-02-13 14:54:09.636130000 | 2032-02-13 14:54:09.636130000 | 2012-02-13 14:54:09.636130000 |
+-------------------------------+-------------------------------+-------------------------------+
```

Note that you can add several `INTERVAL`s together with the same or different units depending on your goal. For example, let's shift today's date by `10` years, `5` minutes, `32` seconds:

```
SELECT NOW() AS TODAY,
       NOW() + INTERVAL 10 YEARS
             + INTERVAL 5 MINUTES
             + INTERVAL 32 SECONDS AS TEN_YEARS_5_MINS_32_SECS_IN_THE_FUTURE;

+-------------------------------+----------------------------------------+
| today                         | ten_years_5_mins_32_secs_in_the_future |
+-------------------------------+----------------------------------------+
| 2022-02-13 15:00:41.484874000 | 2032-02-13 15:06:13.484874000          |
+-------------------------------+----------------------------------------+
```

With the discussion about `INTERVAL` complete, the following functions can also be used to shift `DATE`s and `TIMESTAMP`s. As a reminder, wherever you see a function argument with a data type of `DATE` or `TIMESTAMP`, you can pass in a text string in the appropriate format: `yyyy-mm-dd` or `yyyy-mm-dd hh:mi:ss.SSSSSS`. And, wherever a function argument indicates an interval expression, you can use the `INTERVAL` keyword, as described above.

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
adddate(TIMESTAMP/DATE,*days*)	TIMESTAMP/ DATE	Returns the first argument with the number of days specified in *days* added to it. If *days* is negative, they are subtracted. Note that *days* can be `INT` or `BIGINT`.
date_add(TIMESTAMP/DATE,*days*)	TIMESTAMP/ DATE	Returns the first argument with the number of days specified in *days* added to it. If *days* is negative, they are subtracted. Note that *days* can be `INT` or `BIGINT`.
date_add(TIMESTAMP/DATE,*interval*)	TIMESTAMP/ DATE	Returns the first argument with *interval* added to it
subdate(TIMESTAMP/DATE,*days*)	TIMESTAMP/ DATE	Returns the first argument with the number of days specified in *days* subtracted from it. If *days* is negative, they are added. Note that *days* can be `INT` or `BIGINT`.
date_sub(TIMESTAMP/DATE,*days*)	TIMESTAMP/ DATE	Returns the first argument with the number of days specified in *days* subtracted from it. If *days* is negative, they are added. Note that *days*

		can be `INT` or `BIGINT`.
`date_sub(TIMESTAMP/DATE,interval)`	`TIMESTAMP/DATE`	Returns the first argument with *`interval`* subtracted from it
`days_add(TIMESTAMP/DATE,days)`	`TIMESTAMP/DATE`	Returns the first argument with the number of days specified in *`days`* added to it.
`days_sub(TIMESTAMP/DATE,days)`	`TIMESTAMP/DATE`	Returns the first argument with the number of days specified in *`days`* subtracted from it.
`next_day(TIMESTAMP/DATE,weekday_name)`	`TIMESTAMP/DATE`	Returns the `DATE/TIMESTAMP` of the day, specified in *`weekday_name`*, that follows the date specified in first argument. Note that *`weekday_name`* can be one of the following: `Sunday`/`Sun`, `Monday`/`Mon`, `Tuesday`/`Tue`, `Wednesday`/`Wed`, `Thursday`/`Thu`, `Friday`/`Fri` or `Saturday`/`Sat`.
`last_day(TIMESTAMP/DATE)`	`TIMESTAMP/DATE`	Returns the last day within the same month as the first argument.

For example, let's add `10` years to today's date:

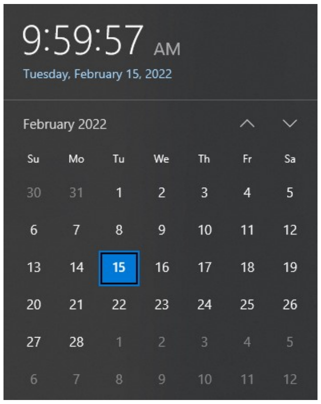
```
SELECT NOW() AS TODAY,
       DATE_ADD(NOW(),INTERVAL 10 YEARS) AS TEN_YEARS_IN_THE_FUTURE;

+-------------------------------+-------------------------------+
| today                         | ten_years_in_the_future       |
+-------------------------------+-------------------------------+
| 2022-02-14 16:38:25.208342000 | 2032-02-14 16:38:25.208342000 |
+-------------------------------+-------------------------------+
```

Now, let's subtract `10` years:

```
SELECT NOW() AS TODAY,
       DATE_SUB(NOW(),INTERVAL 10 YEARS) AS TEN_YEARS_IN_THE_PAST;

+-------------------------------+-------------------------------+
| today                         | ten_years_in_the_past         |
+-------------------------------+-------------------------------+
| 2022-02-14 16:44:34.571261000 | 2012-02-14 16:44:34.571261000 |
+-------------------------------+-------------------------------+
```

Note the two functions at the end of the table above: `NEXT_DAY()` and `LAST_DAY()`. Let's look at some examples. Today is Tuesday, so let's use `NEXT_DAY()` to get the date of this Friday:

```
SELECT NOW() AS TODAY,NEXT_DAY(NOW(),'Friday') AS THIS_FRIDAY

+-------------------------------+-------------------------------+
| today                         | this_friday                   |
+-------------------------------+-------------------------------+
| 2022-02-15 10:00:34.820998000 | 2022-02-18 10:00:34.820998000 |
+-------------------------------+-------------------------------+
```



And that looks correct! Woo-hoo! Next, let's get the last day of this month using `LAST_DAY()`. It better be the 28th or someone's in trouble, Lucy!

```
SELECT NOW() AS TODAY,LAST_DAY(NOW()) AS LAST_DAY_OF_MONTH

+-------------------------------+---------------------+
| today                         | last_day_of_month   |
+-------------------------------+---------------------+
| 2022-02-15 10:03:44.365151000 | 2022-02-28 00:00:00 |
+-------------------------------+---------------------+
```

And, that looks correct as well! Huzzah! Now, the following functions allow you to add or subtract years, months, and weeks:

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
years_add(TIMESTAMP/DATE,*years*)	TIMESTAMP/ DATE	Returns the first argument with the number of years specified in *years* added to it.
years_sub(TIMESTAMP/DATE,*years*)	TIMESTAMP/ DATE	Returns the first argument with the number of years specified in *years* subtracted from it.
add_months(TIMESTAMP/DATE,*months*)	TIMESTAMP/ DATE	Returns the first argument with the number of months specified in *months* added to it. If *months* is negative, they are subtracted. Note that *months* can be INT or BIGINT.
months_add(TIMESTAMP/DATE,*months*)	TIMESTAMP/ DATE	Returns the first argument with the number of months specified in *months* added to it. If *months* is negative, they are subtracted. Note that *months* can be INT or BIGINT.
weeks_add(TIMESTAMP/DATE,*weeks*)	TIMESTAMP/ DATE	Returns the first argument with the number of weeks specified in *weeks* added to it.
weeks_sub(TIMESTAMP/DATE,*weeks*)	TIMESTAMP/ DATE	Returns the first argument with the number of weeks specified in *weeks* subtracted from it.

Let's add 10 years worth of months to today's date:

```
SELECT NOW() AS TODAY,
       ADD_MONTHS(NOW(),120) AS TEN_YEARS_IN_THE_FUTURE;

+-------------------------------+-------------------------------+
| today                         | ten_years_in_the_future       |
+-------------------------------+-------------------------------+
| 2022-02-14 17:29:50.284031000 | 2032-02-14 17:29:50.284031000 |
+-------------------------------+-------------------------------+
```

The following functions are specific to the TIMESTAMP data type since they involve the shifting of time components rather than pure date components:

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
hours_add(TIMESTAMP,*hours*)	TIMESTAMP	Returns the first argument with the number of hours specified in *hours* added to it.
hours_sub(TIMESTAMP,*hours*)	TIMESTAMP	Returns the first argument with the number of hours specified in *hours* subtracted from it.
minutes_add(TIMESTAMP,*minutes*)	TIMESTAMP	Returns the first argument with the number of minutes specified in *minutes* added to it.
minutes_sub(TIMESTAMP,*minutes*)	TIMESTAMP	Returns the first argument with the number of minutes specified in *minutes* subtracted from it.
seconds_add(TIMESTAMP,*seconds*)	TIMESTAMP	Returns the first argument with the number of seconds specified in *seconds* added to it.
seconds_sub(TIMESTAMP,*seconds*)	TIMESTAMP	Returns the first argument with the number of seconds specified in *seconds* subtracted from it.
milliseconds_add(TIMESTAMP,*milliseconds*)	TIMESTAMP	Returns the first argument with the number of milliseconds specified in *milliseconds* added to it.
milliseconds_sub(TIMESTAMP,*milliseconds*)	TIMESTAMP	Returns the first argument with the number of milliseconds specified in *milliseconds* subtracted from it.
microseconds_add(TIMESTAMP,*microseconds*)	TIMESTAMP	Returns the first argument with the number of microseconds

		specified in *microseconds* added to it.
microseconds_sub(TIMESTAMP,*microseconds*)	TIMESTAMP	Returns the first argument with the number of microseconds specified in *microseconds* subtracted from it.
nanoseconds_add(TIMESTAMP,*nanoseconds*)	TIMESTAMP	Returns the first argument with the number of nanoseconds specified in *nanoseconds* added to it.
nanoseconds_sub(TIMESTAMP,*nanoseconds*)	TIMESTAMP	Returns the first argument with the number of nanoseconds specified in *nanoseconds* subtracted from it.

Let's add 1 hour 5 minutes to the current time:

```
SELECT NOW() AS TODAY,
       HOURS_ADD(
                MINUTES_ADD(NOW(),5),
               1) AS ONE_HOUR_FIVE_MINUTES_FROM_NOW;

+-------------------------------+--------------------------------+
| today                         | one_hour_five_minutes_from_now |
+-------------------------------+--------------------------------+
| 2022-02-15 09:53:29.243206000 | 2022-02-15 10:58:29.243206000  |
+-------------------------------+--------------------------------+
```

Finally, let's look at the last three functions:

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
to_date(TIMESTAMP)	STRING	Returns the TIMESTAMP formatted as a STRING.
utc_timestamp()	TIMESTAMP	Returns the current date and time, similar to NOW(), but shifted to the UTC timezone.
from_utc_timestamp(TIMESTAMP,*timezone*)	TIMESTAMP	Converts a UTC timestamp by shifting it to the specified timezone in *timezone*. Please see the documentation for more on the format of *timezone*.

Despite the name, the TO_DATE() function does **not** convert a STRING into a DATE, but just returns the date portion of a TIMESTAMP in yyyy-mm-dd format, like this:

```
SELECT NOW() AS TODAY,TO_DATE(NOW()) AS NICE_LOOKING_TODAY

+-------------------------------+--------------------+
| today                         | nice_looking_today |
+-------------------------------+--------------------+
| 2022-02-15 10:16:39.950483000 | 2022-02-15         |
+-------------------------------+--------------------+
```

The UTC_TIMESTAMP() function returns a TIMESTAMP of the current date and time shifted to the UTC timezone. Since the author's laptop is located on the east coast of the United States (although the whereabouts of the author himself is unknown…probably at the local donut shoppe), we should see a difference of 5 hours between NOW() and UTC_TIMESTAMP():

```
SELECT NOW() AS TODAY,UTC_TIMESTAMP() AS UTC_TODAY

+-------------------------------+-------------------------------+
| today                         | utc_today                     |
+-------------------------------+-------------------------------+
| 2022-02-15 10:20:36.732856000 | 2022-02-15 15:20:36.732856000 |
+-------------------------------+-------------------------------+
```

Now, we can shift a UTC_TIMESTAMP() value to any timezone we desire. For example, let's take the UTC_TODAY value from above and shift it back to Eastern Standard Time (EST):

```
SELECT FROM_UTC_TIMESTAMP('2022-02-15 15:20:36.732856000','EST') AS TODAY

+-------------------------------+
| today                         |
+-------------------------------+
| 2022-02-15 10:20:36.732856000 |
+-------------------------------+
```

And, this value matches the value above.

DATE/TIMESTAMP Functions Parade

Finally, we end this chapter with a complete list of DATE- and TIMESTAMP-related functions for your dining and dancing pleasure.

DATE/TIMESTAMP FUNCTIONS		
Function	**Return Type**	**Description**
cast(*expr* as *type* format *pattern*)	TIMESTAMP/ DATE	Returns *expr* casted to the data type DATE or TIMESTAMP based on the format pattern specified in the FORMAT Clause.
from_timestamp(*timestamp/string*, *output pattern*)	STRING	Converts a TIMESTAMP to a STRING based on the provided output pattern. If the first argument is already a TIMESTAMP, the output pattern is applied to create the desired formatted string. If the first argument is a STRING, it must be in the appropriate TIMESTAMP format yyyy-mm-dd hh:mi:ss.SSSSSS. It will be converted internally to a TIMESTAMP and then the ouput pattern will be applied to create the desired formatted string.
from_unixtime(*unixtime*,*output pattern*)	STRING	The first argument contains a number of seconds since Unix Epoch and the second argument specified the desired output format.
to_timestamp(*date_string*,*input pattern*)	TIMESTAMP	Converts a *date_string* using the provided *input pattern* into a TIMESTAMP.
current_date()	DATE	Returns the current date as a DATE data type.
current_timestamp()	TIMESTAMP	Returns the current date and time as a TIMESTAMP data type.
now()	TIMESTAMP	Returns the current date and time as a TIMESTAMP data type.
timeofday()	STRING	Returns the current date and time as a STRING.
extract(TIMESTAMP/DATE,*unit*) **extract(*unit* from TIMESTAMP/DATE)**	BIGINT	Returns the requested ***unit*** from a TIMESTAMP or DATE data type.
date_part(*unit*,TIMESTAMP/DATE)	BIGINT	Returns the requested ***unit*** from a TIMESTAMP or DATE data type.
day(TIMESTAMP/DATE)	INT	Returns the day number between 1 and 31.
dayofmonth(TIMESTAMP/DATE)	INT	Returns the day number between 1 and 31.
dayname(TIMESTAMP/DATE)	STRING	Returns the name of the day between Sunday to Saturday.
dayofweek(TIMESTAMP/DATE)	INT	Returns the day value and ranges between 1 (Sunday) and 7 (Saturday).
dayofyear(TIMESTAMP/DATE)	INT	Returns the day of the year value and ranges between 1 and 366.
hour(TIMESTAMP)	INT	Returns the hours value.
millisecond(TIMESTAMP)	INT	Returns the milliseconds value.
minute(TIMESTAMP)	INT	Returns the minutes value.
month(TIMESTAMP/DATE)	INT	Returns the month value.
monthname(TIMESTAMP/DATE)	STRING	Returns the month name between January and December.
quarter(TIMESTAMP/DATE)	INT	Returns the quarter value 1 (Jan/Feb/Mar), 2 (Apr/May/Jun), 3 (Jul/Aug/Sep) or 4 (Oct/Nov/Dec).
second(TIMESTAMP)	INT	Returns the seconds value.
week(TIMESTAMP/DATE)	INT	Returns the week of the year between 1 and 53.
weekofyear(TIMESTAMP/DATE)	INT	Returns the week of the year between 1 and 53.
year(TIMESTAMP/DATE)	INT	Returns the year value.
date_trunc(*unit*,TIMESTAMP/DATE)	TIMESTAMP/ DATE	Truncates the TIMESTAMP or DATE truncated to requested ***unit***. The ***unit*** can be one of the following: ▪ MICROSECONDS – Rounds a TIMESTAMP to the microsecond. ▪ MILLISECONDS – Rounds a TIMESTAMP to the millisecond. ▪ SECOND – Rounds a TIMESTAMP to the second. ▪ MINUTE – Rounds a TIMESTAMP to the minute. ▪ HOUR – Rounds a TIMESTAMP to the hour. ▪ DAY – Rounds a DATE/TIMESTAMP to the day. ▪ WEEK – Rounds a DATE/TIMESTAMP to the week.

		▪ `MONTH` – Rounds a `DATE`/`TIMESTAMP` to the month. ▪ `YEAR` – Rounds a `DATE`/`TIMESTAMP` to the year. ▪ `DECADE` – Rounds a `DATE`/`TIMESTAMP` to the decade. ▪ `CENTURY` – Rounds a `DATE`/`TIMESTAMP` to the century. ▪ `MILLENNIUM` – Rounds a `DATE`/`TIMESTAMP` to the millennium.
trunc(TIMESTAMP/DATE,*unit*)	TIMESTAMP/ DATE	Truncates the `TIMESTAMP` or `DATE` truncated to requested ***unit***. The ***unit*** can be one of the following: ▪ `SYYYY`/`YYYY`/`YEAR`/`SYEAR`/ `YYY`/`YY`/`Y` – Rounds a `DATE`/`TIMESTAMP` to the year. ▪ `Q` – Rounds a `DATE`/`TIMESTAMP` to the quarter. ▪ `MONTH`/`MON`/`MM`/`RM` – Rounds a `DATE`/`TIMESTAMP` to the month. ▪ `WW` – Rounds a `DATE`/`TIMESTAMP` to the week. ▪ `DDD`/`DD`/`J` – Rounds a `DATE`/`TIMESTAMP` to the day. ▪ `DAY`/`DY`/`D` – Rounds a `DATE`/`TIMESTAMP` to the starting day of the week. ▪ `HH`/`HH12`/`HH24` – Rounds a `TIMESTAMP` to the hour. ▪ `MI` – Rounds a `TIMESTAMP` to the minute.
DATE_CMP(DATE *dt1*,DATE *dt2*)	INT	Returns the following: $\begin{cases} -1, & if\ dt1 < dt2 \\ 0, & if\ dt1 = dt2 \\ +1, & if\ dt1 > dt2 \end{cases}$ `NULL` will be returned if either argument is `NULL`.
TIMESTAMP_CMP(TIMESTAMP *ts1*, TIMESTAMP *ts2*)	INT	Returns the following: $\begin{cases} -1, & if\ ts1 < ts2 \\ 0, & if\ ts1 = ts2 \\ +1, & if\ ts1 > ts2 \end{cases}$ `NULL` will be returned if either argument is `NULL`.
months_between(TIMESTAMP/DATE, TIMESTAMP/DATE)	DOUBLE	Returns the number of full months between the two dates as a floating-point value. Note that the time portion is ignored in the calculation. The result will be negative if the first argument occurs prior to the second argument.
int_months_between(TIMESTAMP/DATE, TIMESTAMP/DATE)	INT	Similar to `MONTHS_BETWEEN()`, but returns the `FLOOR()` of the computed value. The result will be negative if the first argument occurs prior to the second argument.
datediff(TIMESTAMP/DATE,TIMESTAMP/DATE)	INT	Computes the number of days **between** the two arguments. Note that if you want the total number of days between the two dates, you must add one to the resulting value. The result will be negative is the first argument occurs prior to the second argument.
adddate(TIMESTAMP/DATE,*days*)	TIMESTAMP/ DATE	Returns the first argument with the number of days specified in *`days`* added to it. If *`days`* is negative, they are subtracted. Note that days can be `INT` or `BIGINT`.
date_add(TIMESTAMP/DATE,*days*)	TIMESTAMP/ DATE	Returns the first argument with the number of days specified in *`days`* added to it. If *`days`* is negative, they are subtracted. Note that days can be `INT` or `BIGINT`.
date_add(TIMESTAMP/DATE,*interval*)	TIMESTAMP/ DATE	Returns the first argument with *`interval`* added to it
subdate(TIMESTAMP/DATE,*days*)	TIMESTAMP/ DATE	Returns the first argument with the number of days specified in *`days`* subtracted from it. If *`days`* is negative, they are added. Note that *`days`* can be `INT` or `BIGINT`.
date_sub(TIMESTAMP/DATE,*days*)	TIMESTAMP/ DATE	Returns the first argument with the number of days specified in *`days`* subtracted from it. If *`days`* is negative, they are added. Note that *`days`* can be `INT` or `BIGINT`.
date_sub(TIMESTAMP/DATE,*interval*)	TIMESTAMP/ DATE	Returns the first argument with *`interval`* subtracted from it
days_add(TIMESTAMP/DATE,*days*)	TIMESTAMP/ DATE	Returns the first argument with the number of days specified in *`days`* added to it.
days_sub(TIMESTAMP/DATE,*days*)	TIMESTAMP/ DATE	Returns the first argument with the number of days specified in *`days`* subtracted from it.
next_day(TIMESTAMP/DATE,*weekday_name*)	TIMESTAMP/ DATE	Returns the `DATE/TIMESTAMP` of the day, specified in *`weekday_name`*, that follows the date specified in first argument. Note that *`weekday_name`* can be one of the following: `Sunday`/`Sun`, `Monday`/`Mon`, `Tuesday`/`Tue`, `Wednesday`/`Wed`, `Thursday`/`Thu`, `Friday`/`Fri` or `Saturday`/`Sat`.
last_day(TIMESTAMP/DATE)	TIMESTAMP/	Returns the last day within the same month as the first argument.

	DATE	
hours_add(TIMESTAMP,*hours*)	TIMESTAMP	Returns the first argument with the number of hours specified in *hours* added to it.
hours_sub(TIMESTAMP,*hours*)	TIMESTAMP	Returns the first argument with the number of hours specified in *hours* subtracted from it.
minutes_add(TIMESTAMP,*minutes*)	TIMESTAMP	Returns the first argument with the number of minutes specified in *minutes* added to it.
minutes_sub(TIMESTAMP,*minutes*)	TIMESTAMP	Returns the first argument with the number of minutes specified in *minutes* subtracted from it.
seconds_add(TIMESTAMP,*seconds*)	TIMESTAMP	Returns the first argument with the number of seconds specified in *seconds* added to it.
seconds_sub(TIMESTAMP,*seconds*)	TIMESTAMP	Returns the first argument with the number of seconds specified in *seconds* subtracted from it.
milliseconds_add(TIMESTAMP,*milliseconds*)	TIMESTAMP	Returns the first argument with the number of milliseconds specified in *milliseconds* added to it.
milliseconds_sub(TIMESTAMP,*milliseconds*)	TIMESTAMP	Returns the first argument with the number of milliseconds specified in *milliseconds* subtracted from it.
microseconds_add(TIMESTAMP,*microseconds*)	TIMESTAMP	Returns the first argument with the number of microseconds specified in *microseconds* added to it.
microseconds_sub(TIMESTAMP,*microseconds*)	TIMESTAMP	Returns the first argument with the number of microseconds specified in *microseconds* subtracted from it.
nanoseconds_add(TIMESTAMP,*nanoseconds*)	TIMESTAMP	Returns the first argument with the number of nanoseconds specified in *nanoseconds* added to it.
nanoseconds_sub(TIMESTAMP,*nanoseconds*)	TIMESTAMP	Returns the first argument with the number of nanoseconds specified in *nanoseconds* subtracted from it.
to_date(TIMESTAMP)	STRING	Returns the TIMESTAMP formatted as a STRING.
utc_timestamp()	TIMESTAMP	Returns the current date and time, similar to NOW(), but shifted to the UTC timezone.
from_utc_timestamp(TIMESTAMP,*timezone*)	TIMESTAMP	Converts a UTC timestamp by shifting it to the specified timezone in *timezone*. Please see the documentation for more on the format of *timezone*.

Chapter 11 – Regular Expressions in ImpalaSQL

There are several additional functions in ImpalaSQL which have Regular Expression support. If you're new to regular expressions, don't worry, we explain them in this chapter. At first glance, a regular expression may look like your cat has walked across the keyboard multiple times whilst dribbling a very tiny basketball, but you'll be an old hat at regular expressions by the end of the chapter.

Not only are Regular Expressions available in ImpalaSQL, but they can be used at the Linux command line and in Linux scripts as well.

Regular Expressions are probably available in your favorite programming languages such as Python, R, SAS, C, C++, C#, PHP… the list just goes on and on.

Regular Expressions are probably available in your favorite text editor such as TextPad, Notepad++, UltraEdit, etc.

Regular Expressions are available in Microsoft Word as well!! *Hit me in the head and call me stupid! Throw some mud on me and call me a pig! I did not know Word could do that!*

In general, Regular Expressions are a good thing to add to your resume and you'll find out why below.

So…What's a Regular Expression?

A Regular Expression is **plain text** made up of **special characters** which indicate the **format of the text** you're trying to search for, replace with, or hack apart. In other words, a regular expression is a **template** used as the search criteria within a text string. Note that not all Regular Expressions will match all text strings you're working with, and you'll most likely need more than one Regular Expression in order to match your text strings. If this is confusing, imagine trying to match all of the different forms of addresses out there (e.g., 1700 Neva Road, 1 Glen Bell Way, etc.) and you'll quickly get the point.

Motivational Example

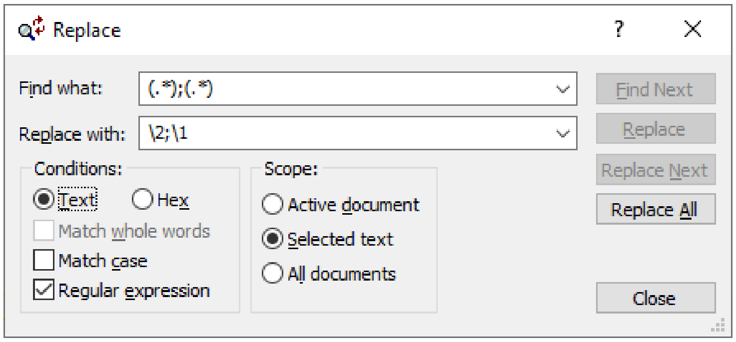
Let's see an example of how you can use Regular Expressions in the text editor TextPad (which has Regular Expression support). You can try the same thing in your favorite text editor, but be aware that your editor may execute regular expressions in a slightly different way to that shown below.

Let's say you downloaded the following data from the *InterWebs*:

```
AllianceBernstein Income Fund, Inc.;ACG
Avenue Income Credit Strategies;ACP
The Adams Express Company;ADX
AllianceBernstein National Municipal Income Fund, Inc;AFB
Apollo Senior Floating Rate Fund, Inc.;AFT
Advent Claymore Convertible Security;AGC
Alpine Global Dynamic Dividend Fund;AGD
Alliance California Municipal Income Fund Inc.;AKP
Alpine Total Dynamic Dividend Fund;AOD
Asia Pacific Fund Inc.;APB
Morgan Stanley Asia-Pacific Fund Inc.;APF
Ares Dynamic Credit Allocation;ARD
BlackRock Senior High Income Fund, Inc.;ARK
ASA Gold and Precious Metals Limited;ASA
Liberty All Star Growth Fund Inc.;ASG
American Strategic Income Portfolio Inc.;ASP
```

Naturally, you'll want to load this data into a database. If you're using Oracle, you can create a SQL*Loader script to load the data into the database. Or, if you're using Microsoft SQL Server, you can create a `bcp` script. But, there's so little data, why would you go through the trouble? Why not just create a few `INSERT` Statements?

Using TextPad, you can quickly swap the data so that the stock symbol appears first and the name after. Using the *Find what* input box, enter in the following regular expression **(.*);(.*)**. In the *Replace with* input box, enter in the following regular expression: **\2;\1**. Note that the period and the asterisk are called *metacharacters* and are two of many. You know, I never metacharacter I didn't like! HA! That's a little joke! Ahem.

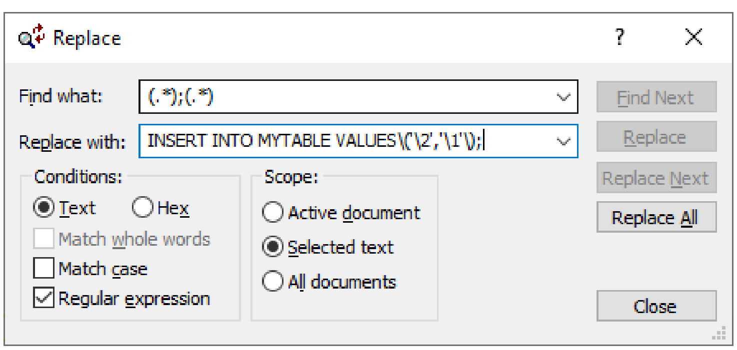


The data now looks like this, the columns swapped:

```
ACG;AllianceBernstein Income Fund, Inc.
ACP;Avenue Income Credit Strategies
ADX;The Adams Express Company
AFB;AllianceBernstein National Municipal Income Fund, Inc
AFT;Apollo Senior Floating Rate Fund, Inc.
AGC;Advent Claymore Convertible Security
AGD;Alpine Global Dynamic Dividend Fund
AKP;Alliance California Municipal Income Fund Inc.
AOD;Alpine Total Dynamic Dividend Fund
APB;Asia Pacific Fund Inc.
APF;Morgan Stanley Asia-Pacific Fund Inc.
ARD;Ares Dynamic Credit Allocation
ARK;BlackRock Senior High Income Fund, Inc.
ASA;ASA Gold and Precious Metals Limited
ASG;Liberty All Star Growth Fund Inc.
ASP;American Strategic Income Portfolio Inc.
```

We can actually go so far as to add the `INSERT INTO` SQL command in the *Replace with* input box:

```
INSERT INTO MYTABLE VALUES\('\2','\1'\);
```




```
INSERT INTO MYTABLE VALUES('ACG','AllianceBernstein Income Fund, Inc.');
INSERT INTO MYTABLE VALUES('ACP','Avenue Income Credit Strategies');
INSERT INTO MYTABLE VALUES('ADX','The Adams Express Company');
INSERT INTO MYTABLE VALUES('AFB','AllianceBernstein National Municipal Income Fund, Inc');
INSERT INTO MYTABLE VALUES('AFT','Apollo Senior Floating Rate Fund, Inc.');
INSERT INTO MYTABLE VALUES('AGC','Advent Claymore Convertible Security');
INSERT INTO MYTABLE VALUES('AGD','Alpine Global Dynamic Dividend Fund');
INSERT INTO MYTABLE VALUES('AKP','Alliance California Municipal Income Fund Inc.');
```

```
INSERT INTO MYTABLE VALUES('AOD','Alpine Total Dynamic Dividend Fund');
INSERT INTO MYTABLE VALUES('APB','Asia Pacific Fund Inc.');
INSERT INTO MYTABLE VALUES('APF','Morgan Stanley Asia-Pacific Fund Inc.');
INSERT INTO MYTABLE VALUES('ARD','Ares Dynamic Credit Allocation');
INSERT INTO MYTABLE VALUES('ARK','BlackRock Senior High Income Fund, Inc.');
INSERT INTO MYTABLE VALUES('ASA','ASA Gold and Precious Metals Limited');
INSERT INTO MYTABLE VALUES('ASG','Liberty All Star Growth Fund Inc.');
INSERT INTO MYTABLE VALUES('ASP','American Strategic Income Portfolio Inc.');
```

Now, there are several things to note about the previous examples:

- The period (`.`) and the asterisk (**`*`**) are *metacharacters* where the period indicates any single character and the asterisk indicates that there are to be zero or more characters. That is, **`.*`** indicates you're searching for zero or more characters. What you're searching for is indicated by the period; how many of those characters are indicated by the asterisk.
- Characters that are **not** metacharacters, such as the semi-colon and the words `INSERT INTO MYTABLE VALUES` play the role of themselves and have no special meaning.
- The parentheses indicate that any matched data will be referenced in the regular expression later on. This is called a *backreference*. In the example above, the first **`(.*)`** captures zero or more characters and is referred to as **`\1`**; the second **`(.*)`** – after the semicolon – is referred to as **`\2`**. It's the semi-colon that separates the two.
- Since parentheses are part of the `INSERT INTO VALUES` syntax, you have to *escape* both parentheses so that your text editor's Regular Expression support doesn't confuse it with a backreference. We talk about escaping further below.
- These concepts are used in Impala SQL's Regular Expression functions, which we explore below.
- Some additional motivational examples follow:
  - `REGEXP_REPLACE(A.CLEAN_ADDRESS_1,'[ ]{2,}',' ')` – replace two or more blanks with a single blank.
  - `REGEXP_REPLACE(WORK_PHONE,'(\d{3})(\d{3})(\d{4})','(\1) \2-\3')` – format the phone number, say, `1234567890` in the format `(123) 456-7890`.
  - `REGEXP_REPLACE(STRING,'\t',' ')` – replace each tab with a space.

You've Seen This Before!

Even though you may be new to Regular Expressions, you're most likely familiar with similar concepts. For example, you've most likely issued a command at the Windows Command Prompt similar to the following:

```
dir bunny*.jpg
```

What does this do? Windows will return a directory listing of all files starting with the word `bunny`, followed by some unknown amount of text (or no text at all), and followed by the extension `.jpg`.

What does the asterisk (`*`) mean in this context? The asterisk is a catchall indicating zero or more letters, numbers, or symbols. You may have also seen something similar when using SQL's `LIKE` Condition:

```
SELECT IMAGE_NAME
 FROM IMAGE_NAME_TABLE
 WHERE IMAGE_NAME LIKE 'bunny%.jpg';
```

What does this do? SQL will display all of the rows where the column `IMAGE_NAME` contains text that starts with the word `bunny`, followed by some unknown amount of text (or no text at all), and followed by the extension `.jpg`.

What does the percent sign (`%`) mean in this context? The percent sign is a catchall, similar to the asterisk above, that indicates zero or more letters, numbers, or symbols.

Recall the underscore (_) matches exactly one character when using the `LIKE` Condition:

```
SELECT IMAGE_NAME
 FROM IMAGE_NAME_TABLE
 WHERE IMAGE_NAME LIKE 'bunny_.jpg';
```

But, there's a problem with both the asterisk (`*`) and the percent sign (`%`). Both symbols are conflating two distinct concepts: the concept of ***the what*** and the concept of ***the how many***; that is, when using the asterisk (`*`) or percent sign (`%`), there's no way we can limit the search criteria to files ending in the numbers, say, `0` to `9` (***the what***): `bunny0.jpg`, `bunny1.jpg`, etc. There's also no way that we can limit the search criteria to, say, look for exactly three numbers in a row (***the how many***), such as `bunny`**`123`**`.jpg` or `bunny`**`789`**`.jpg`. There's also no way that we can limit the search criteria to indicate alternate forms: **`b`**`unny123.jpg`, **`s`**`unny123.jpg`, **`b`**`unny789.jpg`, **`s`**`unny789.jpg`, etc.

Regular Expressions takes the concept of the asterisk (`*`) and the percent sign (`%`) and blows them apart into ***the what*** and ***the how many***. Boom! Now, with that flexibility comes complexity, but you get used to it.

What Does a Regular Expression Involve?

In order to use Regular Expressions to find a match, you need to have two text strings:

- ☐ *`source_string`* – this string contains the text you want to search through using Regular Expressions. For example, this string can contain an address, a movie title, text from a patient's hospital chart, a detailed description of a company, or any other text you need to analyze.
- ☐ *`regex_string`* – this string contains a Regular Expression used to search through the *`source_string`* in order to find a match. This Regular Expression describes the **format** your text string should be in, not the exact text itself.

There are additional parameters you can use when working with ImpalaSQL's Regular Expression functions and we talk about those later on.

When working with Regular Expressions, you need to understand how the regular expression parser sees text. Letters, numbers and symbols (for the most part) are all the same to the regular expression parser.

```
123 Main Street, Apt #A1, AnyCity, AnyState 12345-6789
```

For example, in the text string above,

- ☐ Can you locate text consisting of five numbers in a row?
- ☐ Can you locate text consisting of three numbers in a row?
- ☐ Can you locate text consisting of five numbers in a row, followed by a single dash followed by four numbers in a row?
- ☐ Can you locate text consisting of a house number, followed by a street name, followed by the text `Street`?

Of course you can! The way your brain finds patterns is similar to how Regular Expressions work. In this case, you have to specify the exact Regular Expression format you want to use.

First Look Examples

In this section, we ease you into Regular Expressions by showing you several simple examples and explaining how they work. Note that this section is not supposed to be comprehensive since we'll go into much more detail in the next section.

For this section, we'll be using the Oracle Regular Expression function `REGEXP_INSTR(`*`source_string`*`,` *`regex_string`*`)` which returns the column location within *`source_string`* where the matched *`regex_string`* begins. If the *`regex_string`* is not found, `REGEXP_INSTR()` returns a zero. Now, it may seem subversive to be using an Oracle function in a book about ImpalaSQL, but there isn't a Regular Expression analog available in

ImpalaSQL (`INSTR` is the non-Regular Expression function available in ImpalaSQL). Using this particular Oracle function just makes explaining Regular Expressions much clearer, so we'll go with it for now.
Let's say we have a table called `REGEX_DATA` which contains the following rows of data in the column named `SOURCE_STRING`:

```
123 Main Street, Apt #A1, AnyCity, AnyState 12345-6789
123 Main St, Apt #A1, AnyCity, AnyState 12345-6789
123 Main St, Apt A1, AnyCity, AnyState 12345-6789
123 Main St, Apt A1, AnyCity, AnyState 12345
123A Main St, Apt A1, AnyCity, AnyState 12345
```

Example #1 – Non-Metacharacters Act As Themselves

```
SELECT SOURCE_STRING,
       REGEXP_INSTR(SOURCE_STRING,'Main') AS RESULT
 FROM REGEX_DATA;
```

SOURCE STRING	RESULT
123 **M**ain Street, Apt #A1, AnyCity, AnyState 12345-6789	5
123 **M**ain St, Apt #A1, AnyCity, AnyState 12345-6789	5
123 **M**ain St, Apt A1, AnyCity, AnyState 12345-6789	5
123 **M**ain St, Apt A1, AnyCity, AnyState 12345	5
123A **M**ain St, Apt A1, AnyCity, AnyState 12345	6

As you see, the word `Main` is found where you'd expect it to be found, in column 5 or 6. This indicates that `REGEXP_INSTR()` behaves like `INSTR()` when not using Regular Expressions.

This example solidifies the fact that non-metacharacters act as themselves.

Example #2 – Using the Period (.) and Asterisk (*)

```
                                        the what   the how many
SELECT SOURCE_STRING,
       REGEXP_INSTR(SOURCE_STRING,'.*') AS RESULT
 FROM REGEX_DATA;
```

SOURCE STRING	RESULT
123 Main Street, Apt #A1, AnyCity, AnyState 12345-6789	1
123 Main St, Apt #A1, AnyCity, AnyState 12345-6789	1
123 Main St, Apt A1, AnyCity, AnyState 12345-6789	1
123 Main St, Apt A1, AnyCity, AnyState 12345	1
123A Main St, Apt A1, AnyCity, AnyState 12345	1

In this case, we're using `.*` to find zero, one or more characters in the `SOURCE_STRING`. Where would that occur at? Naturally, it starts at column `1`.

Remember that the period (`.`) indicates any single character and the asterisk (`*`) indicates zero, one or more of the thing directly to the left of it; that is, the asterisk acts on the period. Note that they come in pairs: ***the what*** (on the left) and ***the how many*** (on the right).

Example #3 – Using the Period (.) and Asterisk (*) Again

```
                                         the what   the how many
SELECT SOURCE_STRING,
       REGEXP_INSTR(SOURCE_STRING,'t.*') AS RESULT
 FROM REGEX_DATA;
```

SOURCE STRING	RESULT
123 Main Street, Apt #A1, AnyCity, AnyState 12345-6789	11
123 Main St, Apt #A1, AnyCity, AnyState 12345-6789	11
123 Main St, Apt A1, AnyCity, AnyState 12345-6789	11
123 Main St, Apt A1, AnyCity, AnyState 12345	11
123A Main St, Apt A1, AnyCity, AnyState 12345	12

In this case, we're using **`t.*`** to find text starting with the letter **`t`**, followed by zero, one or more characters in `SOURCE_STRING`. Where would that occur at? Naturally, it starts at the first occurrence of the letter **`t`** and proceeds to the end of the line.

Recall that non-metacharacters act as themselves, so the letter **`t`** is acting as itself and isn't required to be followed by ***the how many***. Remember that the period (**`.`**) indicates any single character and the asterisk (**`*`**) indicates zero, one or more of the thing directly to the left of it; that is, the asterisk acts on the period and not the letter **`t`**.

Example #4 – Finding the Zipcode Using a Character List (**`[]`**)

```
                                          the what              the how many
SELECT SOURCE_STRING,
       REGEXP_INSTR(SOURCE_STRING,'[0123456789]{5}') AS RESULT
 FROM REGEX_DATA;
```

SOURCE STRING	RESULT
123 Main Street, Apt #A1, AnyCity, AnyState **12345**-6789	45
123 Main St, Apt #A1, AnyCity, AnyState **12345**-6789	41
123 Main St, Apt A1, AnyCity, AnyState **12345**-6789	40
123 Main St, Apt A1, AnyCity, AnyState **12345**	40
123A Main St, Apt A1, AnyCity, AnyState **12345**	41

In this case, we're using **`[0123456789]{5}`** to find exactly five (***the how many***) numbers ranging from zero to nine (***the what***). Where would that occur at? The only place where there are five numbers in a row is the five-digit zip code. The brackets (`[]`) indicate a list of individual characters you want to search for (***the what***). We talk more about the brackets later in this chapter.

Example #5 – Finding the Zipcode Using a Character List (**`[]`**) Again

There's a problem with the previous example: only the five-digit zip code is matched, not the nine-digit zip code. How can we match the nine-digit zip code?

```
                                          the what          the how many
SELECT SOURCE_STRING,
       REGEXP_INSTR(SOURCE_STRING,'[0123456789]{5}-?[0123456789]{4}') AS RESULT
 FROM REGEX_DATA;
                                                  the what      the what        the how many
                                                    the how many
```

SOURCE STRING	RESULT
123 Main Street, Apt #A1, AnyCity, AnyState **12345-6789**	45
123 Main St, Apt #A1, AnyCity, AnyState **12345-6789**	41
123 Main St, Apt A1, AnyCity, AnyState **12345-6789**	40
123 Main St, Apt A1, AnyCity, AnyState 12345	**0**
123A Main St, Apt A1, AnyCity, AnyState 12345	**0**

Here, we're using **`[0123456789]{5}`** to find five consecutive numbers from zero to nine and **`[0123456789]{4}`** to find four consecutive numbers from zero to nine. The **`-?`** indicates that we're looking for a dash (***the what***), but either zero or one of them indicated by the question mark (***the how many***). Note in the Regular Expression above we have three pairs of ***the what*** and ***the how many*** in a row building up our desired Regular Expression. At this

point, we've see the asterisk (`*`) indicating zero, one or more of ***the how many*** and the question mark (`?`) indicating zero or one of ***the how many***.

Example #6 – Finding the Zipcode Using a Character List (`[]`) Again Again

But, there's still a problem: the rows with only a five-digit zip code are not found in the previous example (`RESULT=0`) because the Regular Expression is formatted expecting there always to be a nine-digit zip code. But, in our data, only three rows have this exact format. How do we fix this?

the what *the how many*

```
SELECT SOURCE_STRING,
       REGEXP_INSTR(SOURCE_STRING,'[0123456789]{5}(-?[0123456789]{4})?')
                                                                 AS RESULT
 FROM REGEX_DATA;
```

the what *the how many*

SOURCE STRING	RESULT
123 Main Street, Apt #A1, AnyCity, AnyState **12345-6789**	45
123 Main St, Apt #A1, AnyCity, AnyState **12345-6789**	41
123 Main St, Apt A1, AnyCity, AnyState **12345-6789**	40
123 Main St, Apt A1, AnyCity, AnyState **12345**	40
123A Main St, Apt A1, AnyCity, AnyState **12345**	41

Now, from the previous example, we know that **`-?`** means either zero or one dash. Here, by placing parentheses (called a *subexpression group*) around the part of the Regular Expression describing the dash as well as the four-digit zip code, we turn that whole thing into a ***the what***. And after ***the what***, you need to specify ***the how many***. Here, the **`?`** metacharacter (***the how many***) is used to indicate that ***the what*** can occur either once or never. Effectively, the dash and the four digits can either appear or not appear at all.

Take note that you can replace `[0123456789]` with the abbreviated notation `[0-9]`. Similar for letters of the alphabet: `[a-z]` instead of `[abcdefghijklmnopqrstuvwxyz]` and `[A-Z]` instead of `[ABCDEFGHIJKLMN OPQRSTUVWXYZ]`. You can combine all three together like this: `[a-zA-Z0-9]` as well as add any additional characters you see fit, but see the section on Character Classes below before you go crazy, dude!

Example #7 – Finding Alternatives

Take note that our source string contains two alternatives for the word street: `Street` and `St`. How can we find that using Regular Expressions?

the what *the how many*

```
SELECT SOURCE_STRING,
       REGEXP_INSTR(SOURCE_STRING,'(Street|St){1}') AS RESULT
 FROM REGEX_DATA;
```

SOURCE STRING	RESULT
123 Main **Street**, Apt #A1, AnyCity, AnyState 12345-6789	10
123 Main **St**, Apt #A1, AnyCity, AnyState 12345-6789	10
123 Main **St**, Apt A1, AnyCity, AnyState 12345-6789	10
123 Main **St**, Apt A1, AnyCity, AnyState 12345	10
123A Main **St**, Apt A1, AnyCity, AnyState 12345	11

Here, the subexpression group `(Street|St)` makes use of the vertical bar (`|`) to indicate alternative choices to match. You can have many more than two alternatives!

Example #8 – Matching the Address Number

By default, a Regular Expression will look through the entire string to find a match, no matter where the match occurs. Occasionally, you'll want to do that. But, there are times where you need more control. You can force a Regular Expression to peg the match ***to the start*** of the search string or ***to the end*** of the search string. To do this,

you include the **^** or **$** Operators in the Regular Expression. You can combine the two so that the Regular Expression must match the entire search string ***from the start to the end.*** To do this, you include both the **^** and **$** Operators in the Regular Expression.

Now, let's try to match the address number. Recall that we used the Regular Expression **`[0123456789]{5}`** to match the five-digit zip code. We can do something similar for the house number. But, there's a problem: the house number usually starts at the beginning of the address. What happens if there's no house number? We'll accidentally match part of the zip code! Let's use the **^** Operator to help us out:

```
                                             the what         the how many
SELECT SOURCE_STRING,
       REGEXP_INSTR(SOURCE_STRING,'^[0-9]+[A-Z]?') AS RESULT
 FROM REGEX_DATA;                    peg at start  the what         the how many
```

```
SOURCE STRING                                                RESULT
123 Main Street, Apt #A1, AnyCity, AnyState 12345-6789      1
123 Main St, Apt #A1, AnyCity, AnyState 12345-6789          1
123 Main St, Apt A1, AnyCity, AnyState 12345-6789           1
123 Main St, Apt A1, AnyCity, AnyState 12345                1
123A Main St, Apt A1, AnyCity, AnyState 12345               1
```

Here, the caret (**^**) indicates that the match must occur ***starting from column one*** in `SOURCE_STRING`. If not, then no match. Also, instead of specifying an asterisk (*****) or question mark (**?**), I'm using a plus sign (**+**), to indicate ***one or more*** (***the how many***). At this point, we've seen the **?** (zero or one occurrence), the ***** (zero, one or more occurrences) and, finally, **+** (one or more occurrences). Now, I can hear you all sobbing, "*PLEASE…*sniff sniff*…LET THAT BE IT!* "

But, Wait…There's More! Regular Expressions in Depth

In the previous section, we made use of both the period (**.**) and the asterisk (*****) to help form Regular Expressions. These are referred to as *metacharacters* and are two of many. In this section, we'll list all of the metacharacters and briefly explain what they do. In the following sections, we go into more detail.

Regular Expression Metacharacters: *The How Many* Metacharacters

Metacharacter	Description
`*`	matches **zero or more** occurrences of the expression preceding it such as `.*` or `[abc]*`
`+`	matches **one or more** occurrences of the expression preceding it such as `.+` or `[abc]+`
`?`	matches **zero or one** occurrence of the expression preceding it such as `.?` or `[abc]?`
`{m}`	matches **exactly m** occurrences of the expression preceding it such as `a{5}` or `[0123456789]{5}`
`{m,n}`	matches **at least m but not more than n** occurrences of the expression preceding it such as `a{5,8}`
`{m,}`	matches **at least m** occurrences of the expression preceding it such as `a{5,}`

Note that all six entries above can be followed by the Non-Greedy Operator (**?)** in order to indicate that the match should be non-greedy: **`*?`**, **`+?`**, **`??`**, **`{m}?`**, **`{m,n}?`** and **`{m,}?`**. We talk about the Non-Greedy Operator below.

Regular Expression Metacharacters: Anchors

Metacharacter	Description
`^`	indicates that Regular Expressions must match ***starting from the beginning of the string*** such as `^a{5}`
`$`	indicates that Regular Expressions must match ***at the end of the string*** such as `a{5}$`
`^…$`	indicates that Regular Expressions must match ***the entire string*** exactly such as `^a{5}$`

Regular Expression Metacharacters: Back Reference

Metacharacter	Description
`(expression)`	indicates that the expression is to be saved for reference later on in the regular expression such as `(abc)def+`
`\#`	indicates the expression whose value you want returned such as `\1`, `\2`, …, `\9`.

Regular Expression Metacharacters: Character Classes

Rather than using the left and right brackets to specify a list of characters, Regular Expressions have several pre-defined character classes which, hopefully, will save you some typing. Some character classes are listed below.

<table>
<tr><th>Metacharacter</th><th>Description</th></tr>
<tr><td><code>[:alnum:]</code></td><td>Alphanumeric characters and is equivalent to <code>[a-zA-Z0-9]</code></td></tr>
<tr><td><code>[:alpha:]</code></td><td>Alphabetic characters and is equivalent to <code>[a-zA-Z]</code></td></tr>
<tr><td><code>[:digit:]</code></td><td>Digits and is equivalent to <code>[0-9]</code></td></tr>
<tr><td><code>[:space:]</code></td><td>Whitespace characters and is equivalent to <code>[\t\r\n\v\f]</code></td></tr>
<tr><td><code>[:upper:]</code></td><td>Uppercase letters and is equivalent to <code>[A-Z]</code></td></tr>
<tr><td><code>[:lower:]</code></td><td>Lowercase letters and is equivalent to <code>[a-z]</code></td></tr>
<tr><td><code>[:cntrl:]</code></td><td>Control characters and is equivalent to <code>[\x00-\x1F]</code></td></tr>
<tr><td><code>[:blank:]</code></td><td>Space and tab and is equivalent to <code>[\t]</code></td></tr>
<tr><td><code>[:punct:]</code></td><td>Punctuation characters and is equivalent to <code>[!'#S%&'()*+,-./:;<=>?@[/]^_{|}~]</code></td></tr>
</table>

Note that the backslashed characters listed above, called *escape sequences*, have the following meaning:

- ☐ `\t` – indicates a horizontal tab character
- ☐ `\n` – indicates a newline character
- ☐ `\r` – indicates a carriage return character
- ☐ `\f` – indicates a form feed character
- ☐ `\v` – indicates a vertical tab character

Precede a metacharacter with a backslash (****) to indicate that it should be treated as a plain text character and not a metacharacter. For example, to escape the left parenthesis so the Regular Expression parser doesn't assume it's the beginning of a subexpression group, do this: **`\(`**. To escape the backslash itself, precede with a backslash: **`\\`**.

Regular Expression Metacharacters: PERL-Influenced Operators

Metacharacter	Description
`\d`	matches a digit character and is the same as `[[:digit:]]` or `[0-9]`
`\D`	matches a non-digit character and is the same as `[^[:digit:]]` or `[^0-9]`
`\w`	matches a word character and is the same as `[a-zA-Z0-9_]` or `[[:alnum:]_]`
`\W`	matches a non-word character and is the same as `[^a-zA-Z0-9_]` or `[^[:alnum:]_]`
`\s`	matches a whitespace character and is the same as `[ \t\r\n\v\f]` or `[[:space:]]`
`\S`	matches a non-whitespace character and is the same as `[^[:space:]]` or `[^ \t\r\n\v\f]`

The PERL-influenced operators shown above are alternatives to the character classes indicated in the previous section. They tend to make your regular expression a bit more readable.

Note the alternative use of the caret (`^`) symbol in the brackets above. When specified as the first character, the caret (`^`) indicates the negation of the subsequent characters. For example, `[0-9]` is a list of numbers from `0` to `9`, whereas `[^0-9]` indicates all characters **except** `0` to `9`.

Also, note that if you want to specify a dash (`-`) as part of your character list, it must be specified **first** since its natural meaning is to indicate a range. For example, `[0-9]` is a range of characters from 0 to 9, whereas `[-09]` contains the three characters `-`, `0` and `9` only.

String Length Limitations for Regular Expressions

In some implementations of Regular Expressions, the regular expression itself may be limited to a maximum number of characters. For example, in Oracle, you're limited to `512` characters and in SAS, you're limited to `32767`. Please check the documentation for these maximums. In ImpalaSQL, you're limited to the length of a `STRING` data type, or about `1` billion (1GB) characters. If you exceed that, you win a prize!

Regular Expression Functions in ImpalaSQL

In *Chapter 9 – ImpalaSQL Functions Parade*, we mentioned the functions `INSTR`, `SUBSTR` and `REPLACE` are available in ImpalaSQL. As a reminder,

- `INSTR(`*`string_to_search`*`,`*`search_string`*`)` – Returns the position, starting from `1`, of the first occurrence of *`search_string`* within *`string_to_search`*.
- `REPLACE(`*`string_to_update`*`,`*`search_string`*`,`*`replacement_string`*`)` – Returns *`string_to_update`* with all occurrences of *`search_string`* laserbeamed out and replaced by *`replacement_string`*.
- `SUBSTR(`*`string_to_hack`*`,`*`starting_position`*`,`*`length`*`)` – Returns a portion of *`string_to_hack`* starting at *`starting_position`* for a length of `length`.

The following Regular Expressions-aware functions are available in ImpalaSQL:

- `REGEXP_EXTRACT(`*`string_to_search`*`,`*`regex`*`,`*`backref`*`)` – This function returns the portion of *`string_to_search`* indicated by the backreference number *`backref`* given the regular expression *`regex`* containing parenthesized backreferences.
- `REGEXP_REPLACE(`*`string_to_search`*`,`*`regex`*`,`*`replacement_string`*`)` – The functions returns *`string_to_search`* with the matching *`regex`* replaced by *`replacement_string`*.

- ☐ `REGEXP_LIKE(`*`string_to_search,regex,opts`*`)` – This is similar to the `LIKE` Operator useful in the `WHERE` Clause we described in *Chapter 8 – The One About ImpalaSQL*, but with Regular Expression support. This function returns `TRUE` if *`regex`* matches *`string_to_search`*; `FALSE`, if not.

In the example below, we're using `REGEXP_LIKE` to return rows of data containing a full 9-digit zip code containing a dash:

```
SELECT *
 FROM REGEX_DATA
 WHERE REGEXP_LIKE(SOURCE_STRING,'[0-9]{5}-[0-9]{4}');

+----------------------------------------------------------+
| source_string                                            |
+----------------------------------------------------------+
| 123 Main St, Apt A1, AnyCity, AnyState 12345-6789        |
| 123 Main St, Apt #A1, AnyCity, AnyState 12345-6789       |
| 123 Main Street, Apt #A1, AnyCity, AnyState 12345-6789 |
+----------------------------------------------------------+
```

The following example makes use of `REGEXP_REPLACE` to return the house number, street name and street type as separate columns in the `SOURCE_STRING`. Take note that the backreferences, usually indicated by `\1`, `\2`, etc., are themselves escaped with a single backslash in the third argument to the function.

```
SELECT REGEXP_REPLACE(SOURCE_STRING,'([0-9]+[A-Z]?) ([a-zA-Z]+) ((Street|St)+).*',
                                                              '\\1') AS HOUSE_NUMBER,
       REGEXP_REPLACE(SOURCE_STRING,'([0-9]+[A-Z]?) ([a-zA-Z]+) ((Street|St)+).*',
                                                             '\\2') AS STREET_NAME,
       REGEXP_REPLACE(SOURCE_STRING,'([0-9]+[A-Z]?) ([a-zA-Z]+) ((Street|St)+).*',
                                                             '\\3') AS STREET_TYPE
 FROM REGEX_DATA;

+--------------+-------------+-------------+
| house_number | street_name | street_type |
+--------------+-------------+-------------+
| 123          | Main        | St          |
| 123A         | Main        | St          |
| 123          | Main        | St          |
| 123          | Main        | St          |
| 123          | Main        | Street      |
+--------------+-------------+-------------+
```

The next example is similar to the example shown above, but makes use of the `REGEXP_EXTRACT` function, so there's no need to escape the backreferences in the third argument:

```
SELECT REGEXP_EXTRACT(SOURCE_STRING,'([0-9]+[A-Z]?) ([a-zA-Z]+) ((Street|St)+).*',1)
                                                                   AS HOUSE_NUMBER,
       REGEXP_EXTRACT(SOURCE_STRING,'([0-9]+[A-Z]?) ([a-zA-Z]+) ((Street|St)+).*',2)
                                                                   AS STREET_NAME,
       REGEXP_EXTRACT(SOURCE_STRING,'([0-9]+[A-Z]?) ([a-zA-Z]+) ((Street|St)+).*',3)
                                                                   AS STREET_TYPE
 FROM REGEX_DATA;

+--------------+-------------+-------------+
| house_number | street_name | street_type |
+--------------+-------------+-------------+
| 123          | Main        | St          |
| 123A         | Main        | St          |
| 123          | Main        | St          |
| 123          | Main        | St          |
```

```
| 123          | Main        | Street      |
+--------------+-------------+-------------+
```

There are additional operators – above and beyond those shown in *Chapter 8 – The One About ImpalaSQL* – but with Regular Expression support:

- *string_to_search* REGEXP *regex* Operator – This operator is great to use with a lone SELECT Clause to test out regular expressions, but can also be used on the WHERE Clause similar to REGEXP_LIKE, but without the options. For example, to test if there are three consecutive numbers at the beginning of an address, you can code something like this, which returns TRUE:

```
select '123 MAIN STREET' regexp '^[0-9]{3}.*';
```

  - *string_to_search* IREGEXP *regex* Operator – This operator is the case-insensitive version of the REGEXP Operator described above.
  - RLIKE Operator – This operator is a synonym for the REGEXP Operator described above.
  - In the example above, I want to point out that the caret (^) symbol is being used as an **anchor** and **not as negation** to the character list. The caret (^) would need to be moved inside the brackets in the first position for it to negate 0-9: [**^**0-9]. But, this Regular Expression would have a completely different meaning than the intended one, unless that's your intended intention.

Don't Be Greedy, You Dirty Little Ferret!

As we've seen in the examples above, the Regular Expression **.*** means – referring back to my ***the what*** and ***the how many*** blatherings – *match zero, one or more characters*. But, this Regular Expression tends to be a bit of a pig and may match more characters than you intend. This is true of Regular Expressions in general, and the work-around is to use the Non-Greedy Operator (**?**) after ***the how many***. For example, let's capture the word boulevard from the text boulevard ard ard:

```
SELECT REGEXP_EXTRACT('boulevard ard ard','(b.*d).*',1);

+---------------------------------------------------+
| regexp_extract('boulevard ard ard', '(b.*d).*', 1) |
+---------------------------------------------------+
| boulevard ard ard                                 |
+---------------------------------------------------+
```

As you see in the output above, the entire string boulevard ard ard has been captured instead of just the word boulevard. This is because the Regular Expressions parser, by default, searches for the most characters it can; in this case, starting with the letter **b** and running all the way to the final letter **d**. If we change the Regular Expression by tacking on the Non-Greedy Operator (**?**) to ***the how many***, the correct text is returned because less piggyness is happening:

```
SELECT REGEXP_EXTRACT('boulevard ard ard','(b.*?d).*',1);

+----------------------------------------------------+
| regexp_extract('boulevard ard ard', '(b.*?d).*', 1) |
+----------------------------------------------------+
| boulevard                                          |
+----------------------------------------------------+
```

Now, the Non-Greedy Operator (**?**) can be used **after** the following ***the how many*** metacharacters: ***?**, +**?**, ?**?**, {m}**?**, {m,n}**?** and {m,}**?**.

The takeaway is that if your Regular Expressions are capturing way too many characters, try adding the Non-Greedy Operator (**?**) after ***the how many***…fingers crossed!!

REGEXP_* Functions Parade

Finally, below is a full list of the regular expression functions available in ImpalaSQL.

REGEXP_* FUNCTIONS		
Function	**Return Type**	**Description**
regexp_extract(*str*,*re*,*backref*)	STRING	Returns the portion of the string *str* indicated by the back reference number *backref* (1, 2, …) based on the regular expression in *re* containing parenthesized backreferences.
regexp_replace(*str*,*re*,*replstr*)	STRING	Returns *str* with the matching regular expression in *re* replace by the replacement string in *replstr*.
regexp_like(*str*,*re*,*opts*)	BOOLEAN	Returns TRUE if *re* matches the string in *str*; FALSE, if not.
str* REGEXP *re ***str* RLIKE *re***	BOOLEAN	Returns TRUE if *re* matches the string in *str*; FALSE, if not.
str* IREGEXP *re	BOOLEAN	Returns TRUE if *re* matches the string in *str*; FALSE, if not. This is case-insensitive…don't cry.

Chapter 12 – SQL Analytic (Windowing) Functions in ImpalaSQL

ImpalaSQL supports analytic functions, so in this chapter, we'll describe them briefly. If you've never worked with analytic functions before, you'll be pleasantly surprised and, I'm telling you now, once you start working with them, you'll wonder how you got along without them!

In this chapter, we'll be using the universally disliked, wildly inappropriate, much maligned and politically incorrect `FATKIDS` dataset, shown below in all its caloric glory:

```
+-----------+--------+---------------------+--------+--------+
| firstname | gender | birthdate           | height | weight |
+-----------+--------+---------------------+--------+--------+
| ROSEMARY  | F      | 2000-05-08 00:00:00 | 35     | 123    |
| TOMMY     | M      | 1998-12-11 00:00:00 | 78     | 167    |
| BUDDY     | M      | 1998-10-02 00:00:00 | 45     | 189    |
| ALBERT    | M      | 2000-08-02 00:00:00 | 45     | 150    |
| SIMON     | M      | 1999-01-03 00:00:00 | 87     | 256    |
| FARQUAR   | M      | 1998-11-05 00:00:00 | 76     | 198    |
| LAUREN    | F      | 2000-06-10 00:00:00 | 54     | 876    |
+-----------+--------+---------------------+--------+--------+
```

The columns `FIRSTNAME` and `GENDER` are `STRING`s, `BIRTHDATE` is a `TIMESTAMP` and `HEIGHT` and `WEIGHT` are `INT`s.

As a reminder, an *aggregate function* is used (usually) along with a `GROUP BY` Clause to summarize data in a column down to some level such as `FIRSTNAME`, `GENDER`, etc. As you know, there are several aggregate functions available such as `COUNT`, `SUM`, `AVG` and so on. After an aggregate function completes, the resulting rows are usually less than the total number of rows in the incoming table. For example, let's count the number of fat kids by `GENDER`:

```
SELECT GENDER,COUNT(*) AS KID_COUNT
 FROM FATKIDS
 GROUP BY GENDER
 ORDER BY GENDER;

+--------+-----------+
| gender | kid_count |
+--------+-----------+
| F      | 2         |
| M      | 5         |
+--------+-----------+
```

As you see, two rows are returned, one for the males and one for the females.

On the other hand, *analytic functions* are performed on **the results of a SQL query** and, as such, are the last computations made after the SQL query has finished uttering its incantations. What does that mean? Your SQL query may be very complex, limit incoming data with a `WHERE` Clause, summarize data with a `GROUP BY` Clause, limit output rows using the `HAVING` Clause, etc., but all analytic functions are computed off the results of your complex SQL query. In other words, the number of outgoing rows is the same as the number of incoming rows from the completed query as far as analytic functions are concerned.

For example, let's add an additional column to `FATKIDS` containing the total number of males and female (`KID_COUNT`, from above). There are several ways to do this, one by creating a table from the SQL code above and then joining the table `FATKIDS` to it; another is by using the `WITH` Clause, like this:

```
WITH vwTOT AS (
               SELECT GENDER,COUNT(*) AS KID_COUNT
                FROM FATKIDS
                GROUP BY GENDER
               )
SELECT A.FIRSTNAME,A.GENDER,A.BIRTHDATE,A.HEIGHT,A.WEIGHT,
       B.KID_COUNT
 FROM FATKIDS A INNER JOIN vwTOT B
 ON A.GENDER=B.GENDER
 ORDER BY KID_COUNT;
```

```
+-----------+--------+---------------------+--------+--------+-----------+
| firstname | gender | birthdate           | height | weight | kid_count |
+-----------+--------+---------------------+--------+--------+-----------+
| ROSEMARY  | F      | 2000-05-08 00:00:00 | 35     | 123    | 2         |
| LAUREN    | F      | 2000-06-10 00:00:00 | 54     | 876    | 2         |
| SIMON     | M      | 1999-01-03 00:00:00 | 87     | 256    | 5         |
| BUDDY     | M      | 1998-10-02 00:00:00 | 45     | 189    | 5         |
| ALBERT    | M      | 2000-08-02 00:00:00 | 45     | 150    | 5         |
| TOMMY     | M      | 1998-12-11 00:00:00 | 78     | 167    | 5         |
| FARQUAR   | M      | 1998-11-05 00:00:00 | 76     | 198    | 5         |
+-----------+--------+---------------------+--------+--------+-----------+
```

Now, let's try the same thing using the `COUNT` function in an *analytic* sense rather than an *aggregate* sense. The results are exactly the same as above, but you have one line of code, not 10-ish OCD-formatted-space-infused lines:

```
SELECT A.FIRSTNAME,A.GENDER,A.BIRTHDATE,A.HEIGHT,A.WEIGHT,
       COUNT(*) OVER (PARTITION BY A.GENDER) AS KID_COUNT
 FROM FATKIDS A;
```

So, what in the name of all that's holy is going on here? Well, we're using the `COUNT` function in an *analytic* sense, not an *aggregate* sense. How can you tell? The `OVER` Clause is a dead giveaway that you're using `COUNT` (or `SUM`, `AVG`, etc.) in an *analytic* sense! But, regardless in what sense you're using the `COUNT` function (or `SUM`, `AVG`, etc.), the function still performs its reason for living: `COUNT` counts stuff, `SUM` adds stuff, `AVG` has **mean**ing, and so on.

Now that we've established the `COUNT` function used in an *analytic* sense just counts stuff just like in an *aggregate* sense, what's the `PARTITION BY` Clause doing. At least initially, you can think of the `PARTITION BY` Clause as similar to the `GROUP BY` Clause. The results of the SQL query, all seven rows here, are broken apart into genders, one chunk of `2` rows (the girls) and another chunk of `5` rows (the boys). The `COUNT` function will count the rows for the girls and then count the rows for the boys. What comes out is the original seven rows, with the count of the boys set to `5` **for each of the five male rows**, and the count of the girls set to `2` **for each of the two female rows**.

Whenever you see analytic functions in a SQL query, you can initially dismiss them and just concentrate on the query responsible for pulling, limiting, summarizing, etc. the data. In this case, our SQL query is a very simple query against the `FATKIDS` table. Since there are seven rows coming in and we're not using a `GROUP BY` Clause or `WHERE` Clause, there are seven rows coming out. So, what does the `COUNT` function see? In total, exactly seven rows. Although seven rows are provided to the `COUNT` function, the rows will be split up, or *partitioned*, by `GENDER` and the `COUNT` function will do its usual *thang* by counting the number of males and then the number of females.

As a second motivational example, let's create a **running total** of `WEIGHT` by `GENDER`:

```
SELECT A.GENDER,A.FIRSTNAME,A.WEIGHT,
       SUM(A.WEIGHT) OVER (PARTITION BY A.GENDER
                           ORDER BY A.WEIGHT) AS WT_RUN
 FROM FATKIDS A
 ORDER BY A.GENDER,A.WEIGHT;
```

```
+--------+-----------+--------+--------+
| gender | firstname | weight | wt_run |
+--------+-----------+--------+--------+
| F      | ROSEMARY  | 123    | 123    |
| F      | LAUREN    | 876    | 999    |
| M      | ALBERT    | 150    | 150    |
| M      | TOMMY     | 167    | 317    |
| M      | BUDDY     | 189    | 506    |
| M      | FARQUAR   | 198    | 704    |
| M      | SIMON     | 256    | 960    |
+--------+-----------+--------+--------+
```

partition boundary

Just like with our first example, seven rows come in and seven rows go out. In this case, we're using the SUM function in an *analytic* sense (see the OVER Clause?). Just as before, we're partitioning by GENDER, so the SUM function will see the two rows for the girls and the five rows for the boys. What does SUM do? Well, it adds stuff up, and in this case it's adding the WEIGHT. But, we've coded the ORDER BY Clause within the OVER Clause to indicate we want the WEIGHT sorted from lowest to highest. If you look carefully at the column WT_RUN, you'll see that within the males, the weight is being summed up, and the same for the females. Result: ROLLING COUNTS!! SWEET AS!!

Notice something very important: once you move from the female to the male rows, the sum is set to 150 for Albert's row. This is true for all analytic functions. I like to think of the PARTITION BY Clause as setting up a partition boundary – indicated by the horizontal line in the output above – based on the indicated column (such as GENDER). Once you cross a partition boundary, the analytic function resets for the next partition. This means that there's no bleed-over between partitions.

PARTITION BY Clause

The PARTITION BY Clause, also known as the *Query Partition Clause*, breaks up the incoming data into chunks, or partitions. The analytic function being used is passed each partition which acts on that partition's data. Note that re-initialization occurs when crossing each partition boundary preventing one partition's data from affecting another partition.

In general, the syntax looks like this:

***function*(…) OVER (PARTITION BY *col1*,*col2*,…)**

The *function* above can be one of the familiar aggregate functions COUNT, SUM, AVG, MIN and MAX, but can also be one of the new analytic-specific functions ROW_NUMBER, LEAD, LAG, FIRST_VALUE, LAST_VALUE, RANK, DENSE_RANK, NTILE, CUME_DIST and PERCENT_RANK. We discuss these new functions further below.

Note that, unlike for aggregate functions, the DISTINCT keyword is not allowed in ImpalaSQL functions being used in an analytic sense at the present time.

Note that the PARTITION BY Clause is not always necessary. In the example below, we can count the total number of rows across the incoming data, rather than partitioning by, say, GENDER. But, you still need to provide the OVER Clause with no columns specified, shown as an empty set of parentheses:

```
SELECT A.FIRSTNAME,A.GENDER,A.BIRTHDATE,A.HEIGHT,A.WEIGHT,
       COUNT(*) OVER () AS TOTAL_ROWS
 FROM FATKIDS A;
```

```
+-----------+--------+---------------------+--------+--------+------------+
| firstname | gender | birthdate           | height | weight | total_rows |
+-----------+--------+---------------------+--------+--------+------------+
| FARQUAR   | M      | 1998-11-05 00:00:00 | 76     | 198    | 7          |
| ALBERT    | M      | 2000-08-02 00:00:00 | 45     | 150    | 7          |
```

```
| BUDDY      | M      | 1998-10-02 00:00:00 | 45     | 189    | 7          |
| TOMMY      | M      | 1998-12-11 00:00:00 | 78     | 167    | 7          |
| LAUREN     | F      | 2000-06-10 00:00:00 | 54     | 876    | 7          |
| SIMON      | M      | 1999-01-03 00:00:00 | 87     | 256    | 7          |
| ROSEMARY   | F      | 2000-05-08 00:00:00 | 35     | 123    | 7          |
+-----------+--------+---------------------+--------+--------+------------+
```

As shown in the syntax on the previous page, the `PARTITION BY` Clause accepts several columns. In this next example, we partition by both the gender as well as birth year (pulled from the `BIRTHDATE` using the `EXTRACT` function):

```
SELECT A.FIRSTNAME,A.GENDER,A.BIRTHDATE,A.HEIGHT,A.WEIGHT,
       COUNT(*) OVER (PARTITION BY GENDER,EXTRACT(A.BIRTHDATE,'YEAR'))
                                                          AS TOTAL_ROWS
 FROM FATKIDS A;
```

```
+-----------+--------+---------------------+--------+--------+------------+
| firstname | gender | birthdate           | height | weight | total_rows |
+-----------+--------+---------------------+--------+--------+------------+
| LAUREN     | F      | 2000-06-10 00:00:00 | 54     | 876    | 2          |
| ROSEMARY   | F      | 2000-05-08 00:00:00 | 35     | 123    | 2          |
| TOMMY      | M      | 1998-12-11 00:00:00 | 78     | 167    | 3          |
| FARQUAR    | M      | 1998-11-05 00:00:00 | 76     | 198    | 3          |
| BUDDY      | M      | 1998-10-02 00:00:00 | 45     | 189    | 3          |
| SIMON      | M      | 1999-01-03 00:00:00 | 87     | 256    | 1          |
| ALBERT     | M      | 2000-08-02 00:00:00 | 45     | 150    | 1          |
+-----------+--------+---------------------+--------+--------+------------+
```

partition boundaries

Notice in the above, there are several partition boundaries based on the combination of gender and birth year (`F`/`2000`, `M`/`1998`, `M`/`1999` and `M`/`2000`). Each time a partition boundary is crossed, the analytic function is reset.

ORDER BY Clause

The `ORDER BY` Clause imposes an ordering on the incoming data provided to an analytic function. This clause is required for some analytic functions (`ROW_NUMBER`, `LEAD`, `LAG`, etc.), doesn't really make sense on others (`COUNT`), and may be necessary for some functions depending on your desired results (`SUM`, etc.).

In general, the syntax looks like this:

```
function(…) OVER ( … ORDER BY col1,col2, … )
```

Although it's not indicated in the syntax above, the `PARTITION BY` Clause can also be provided, but it's not always necessary depending on your desired results.

One of the new analytic functions is `ROW_NUMBER` which creates an ever-increasing integral value starting at one. You can use this function with the `PARTITION BY` Clause and once it crosses a partition boundary, the numbering restarts at one. The `ORDER BY` Clause is required in order to enforce some order on the data for the numbering. For example, let's assign a row number on the `FATKIDS` table based on the order of the `FIRSTNAME`:

```
SELECT A.*,
       ROW_NUMBER() OVER (ORDER BY A.FIRSTNAME) AS RNUM
 FROM FATKIDS A;
```

```
+-----------+--------+---------------------+--------+--------+------+
| firstname | gender | birthdate           | height | weight | rnum |
+-----------+--------+---------------------+--------+--------+------+
| ALBERT     | M      | 2000-08-02 00:00:00 | 45     | 150    | 1    |
| BUDDY      | M      | 1998-10-02 00:00:00 | 45     | 189    | 2    |
```

```
| FARQUAR   | M      | 1998-11-05 00:00:00 | 76     | 198    | 3    |
| LAUREN    | F      | 2000-06-10 00:00:00 | 54     | 876    | 4    |
| ROSEMARY  | F      | 2000-05-08 00:00:00 | 35     | 123    | 5    |
| SIMON     | M      | 1999-01-03 00:00:00 | 87     | 256    | 6    |
| TOMMY     | M      | 1998-12-11 00:00:00 | 78     | 167    | 7    |
+-----------+--------+---------------------+--------+--------+------+
```

As you see, the column RNUM starts at 1 and increases to 7 based on the ascending order of the FIRSTNAME.

Next, let's create a row number based on the order of the FIRSTNAME, but also by partitioning by GENDER:

```
SELECT A.*,
       ROW_NUMBER() OVER (PARTITION BY A.GENDER
                          ORDER BY A.FIRSTNAME) AS RNUM
 FROM FATKIDS A;
```

```
+-----------+--------+---------------------+--------+--------+------+
| firstname | gender | birthdate           | height | weight | rnum |
+-----------+--------+---------------------+--------+--------+------+  partition
| LAUREN    | F      | 2000-06-10 00:00:00 | 54     | 876    | 1    |  boundary
| ROSEMARY  | F      | 2000-05-08 00:00:00 | 35     | 123    | 2    |
| ALBERT    | M      | 2000-08-02 00:00:00 | 45     | 150    | 1    |
| BUDDY     | M      | 1998-10-02 00:00:00 | 45     | 189    | 2    |
| FARQUAR   | M      | 1998-11-05 00:00:00 | 76     | 198    | 3    |
| SIMON     | M      | 1999-01-03 00:00:00 | 87     | 256    | 4    |
| TOMMY     | M      | 1998-12-11 00:00:00 | 78     | 167    | 5    |
+-----------+--------+---------------------+--------+--------+------+
```

Note that the numbering restarts at 1 after crossing the partition boundary based on GENDER.

The LEAD analytic function allows you to peak ahead a number of rows while the LAG analytic function allows you to look back a number of rows. Naturally, the ORDER BY Clause is required to put an order on the data so that **look ahead** and **look back** have meaning. Both LEAD and LAG are based on the idea of a *current row*. When accessing tables using SQL, no one really thinks about an individual row, but when using LEAD and LAG, you should. For example, below is the FATKIDS table and let's assume that the *current row* is TOMMY's row:

```
+--------+-----------+--------+
| gender | firstname | weight |
+--------+-----------+--------+
| F      | ROSEMARY  | 123    |
| F      | LAUREN    | 876    |
| M      | ALBERT    | 150    |  <---- LAG 1 row
| M      | TOMMY     | 167    |  <---- Current row
| M      | BUDDY     | 189    |
| M      | FARQUAR   | 198    |  <---- LEAD 2 rows
| M      | SIMON     | 256    |
+--------+-----------+--------+
```

With the data ordered as shown above, lagging back one row from the current row is ALBERT's row. Leading forward two rows from the current row is FARQUAR's row. Here's the syntax for both LEAD and LAG:

```
LEAD(column-name,nbr-rows-to-lead,default-value) OVER (…)
LAG(column-name,nbr-rows-to-lag,default-value) OVER (…)
```

Note that, as stated above, the ORDER BY Clause is required, but the PARTITION BY Clause is not. The *default-value* is returned in the following situations:

1. The partition boundary has been crossed.
2. You lead off the **bottom** of the table into the nether region.

3. You lag off the **top** of the table into the toupee region.

For example, let's use the `LEAD` function to pull the value of `WEIGHT` one row forward from the current row, and use the `LAG` function to pull the `WEIGHT` two rows back from the current row:

```
SELECT A.FIRSTNAME,A.WEIGHT,
       LEAD(A.WEIGHT,1,-1) OVER (ORDER BY A.WEIGHT) AS LEAD_1_WT,
       LAG(A.WEIGHT,2,-1)  OVER (ORDER BY A.WEIGHT) AS LAG_2_WT
 FROM FATKIDS A
 ORDER BY A.WEIGHT;
```

```
+-----------+--------+-----------+----------+
| firstname | weight | lead_1_wt | lag_2_wt |
+-----------+--------+-----------+----------+
| ROSEMARY  | 123    | 150       | -1       |
| ALBERT    | 150    | 167       | -1       |
| TOMMY     | 167    | 189       | 123      |
| BUDDY     | 189    | 198       | 150      |
| FARQUAR   | 198    | 256       | 167      |
| SIMON     | 256    | 876       | 189      |
| LAUREN    | 876    | -1        | 198      |
+-----------+--------+-----------+----------+
```

Note that we're not using the `PARTITION BY` Clause here, but we're ordering the data by ascending `WEIGHT` using the `ORDER BY` Clause. Remember: you must think in terms of the current row in order to determine the lag of two rows and the lead of one row. For instance, on `FARQUAR`'s row, if we lag back two rows, we're on `TOMMY`'s row with a `WEIGHT` of 167. The value 167 is then placed in the column `LAG_2_WT` on `FARQUAR`'s row!! And, if we lead by one row, we're on `SIMON`'s row with a `WEIGHT` of 256. The value 256 is then placed in the column `LEAD_1_WT` on `FARQUAR`'s row!!

Note that, if we lag back two rows when on `ROSEMARY`'s row or `ALBERT`'s row, we're extending above the table's upper boundary, which is why you see the default value of `-1` for `LAG_2_WT` on those two rows. Similar for `LAUREN`'s row when extending beyond the table's lower boundary.

As another example, let's determine the next heaviest weight and the previous lightest weight within `GENDER` using `LEAD` and `LAG`.

```
SELECT A.FIRSTNAME,A.GENDER,A.WEIGHT,
       LEAD(A.WEIGHT,1,-1) OVER (PARTITION BY A.GENDER
                                 ORDER BY A.WEIGHT) AS LEAD_1_WT,
       LAG(A.WEIGHT,2,-1) OVER (PARTITION BY A.GENDER
                                ORDER BY A.WEIGHT) AS LAG_2_WT
 FROM FATKIDS A
 ORDER BY A.GENDER,A.WEIGHT;
```

```
+-----------+--------+--------+-----------+----------+
| firstname | gender | weight | lead_1_wt | lag_2_wt |
+-----------+--------+--------+-----------+----------+
| ROSEMARY  | F      | 123    | 876       | -1       |
| LAUREN    | F      | 876    | -1        | -1       |
| ALBERT    | M      | 150    | 167       | -1       |
| TOMMY     | M      | 167    | 189       | -1       |
| BUDDY     | M      | 189    | 198       | 150      |
| FARQUAR   | M      | 198    | 256       | 167      |
| SIMON     | M      | 256    | -1        | 189      |
+-----------+--------+--------+-----------+----------+
```

partition boundary

Note that on ROSEMARY's row, her WEIGHT is 123, but when leading one row, you get LAUREN's row of 876. The value of 876 appears in the column LEAD_1_WT on ROSEMARY's row!! Note that when LAUREN's row is the current row, both LEAD_1_WT and LAG_2_WT are set to the default value of -1 since leading by one row crosses the partition boundary from the female rows to the male rows, and lagging by two rows breaches the upper boundary of the table.

Two other analytic functions that require the ORDER BY Clause are RANK and DENSE_RANK. You can think of these functions as the rank order of runners passing the finish line. If two runners finish in first place, they're both ranked as 1 and the next runner to finish is ranked as 3, not 2. This is how the RANK analytic function works. The DENSE_RANK analytic function is the same as RANK except values are not skipped. If two runners finish in first place, they're both ranked as 1 and the next runner to finish is ranked 2, not 3. Hence, the use of the word **dense** in the function's name

Here's the syntax for both analytic functions. Take note that no parameter is required since the ORDER BY Clause specifies how the data is to be ordered. Also, just like with the analytic functions above, the values are reset when crossing a partition boundary.

```
RANK() OVER ( … ORDER BY col1,col2, … )
DENSE_RANK() OVER ( … ORDER BY col1,col2, … )
```

Note that the PARTITION BY Clause is optional.

For example, let's create a ranking of ascending HEIGHT within GENDER using both RANK and DENSE_RANK.

```
SELECT A.FIRSTNAME,A.GENDER,A.HEIGHT,
       RANK() OVER (PARTITION BY A.GENDER
                    ORDER BY A.HEIGHT) AS HT_RANK,
       DENSE_RANK() OVER (PARTITION BY A.GENDER
                          ORDER BY A.HEIGHT) AS HT_DENSERANK
 FROM FATKIDS A
 ORDER BY A.GENDER,A.HEIGHT;
```

```
+-----------+--------+--------+---------+--------------+
| firstname | gender | height | ht_rank | ht_denserank |
+-----------+--------+--------+---------+--------------+
| ROSEMARY  | F      | 35     | 1       | 1            |
| LAUREN    | F      | 54     | 2       | 2            |
| ALBERT    | M      | 45     | 1       | 1            |
| BUDDY     | M      | 45     | 1       | 1            |
| FARQUAR   | M      | 76     | 3       | 2            |
| TOMMY     | M      | 78     | 4       | 3            |
| SIMON     | M      | 87     | 5       | 4            |
+-----------+--------+--------+---------+--------------+
```

partition boundary

In the output above, you'll notice that the boys are ranked 1, 1, 3, 4, 5 under the HT_RANK column whereas they're ranked 1, 1, 2, 3, 4 under the HT_DENSERANK column.

The FIRST_VALUE analytic function returns the first value within a partition boundary whereas the LAST_VALUE analytic function returns the last value within a partition boundary. Naturally, the ORDER BY Clause is required since an ordering must be placed on the data in order to determine what's first and what's last. Note that the PARTITION BY Clause is not required, and these two analytic functions will return the first and last values within the entire table based on the desired ordering.

Here's the syntax for the FIRST_VALUE and LAST_VALUE analytic functions:

```
FIRST_VALUE(column-name) OVER ( … ORDER BY col1, col2, … )
LAST_VALUE(column-name) OVER ( … ORDER BY col1, col2, … )
```

For example, let's retrieve the names of the heaviest and lightest kids by `GENDER`:

```
SELECT A.FIRSTNAME,A.GENDER,A.WEIGHT,
       FIRST_VALUE(A.FIRSTNAME) OVER (PARTITION BY A.GENDER
                                     ORDER BY A.WEIGHT) AS LT_CHILD,
       LAST_VALUE(A.FIRSTNAME) OVER (PARTITION BY A.GENDER
                                    ORDER BY A.WEIGHT) AS HV_CHILD
 FROM FATKIDS A
 ORDER BY A.GENDER,A.WEIGHT;
```

```
+-----------+--------+--------+----------+----------+
| firstname | gender | weight | lt_child | hv_child |
+-----------+--------+--------+----------+----------+
| ROSEMARY  | F      | 123    | ROSEMARY | ROSEMARY |
| LAUREN    | F      | 876    | ROSEMARY | LAUREN   |
| ALBERT    | M      | 150    | ALBERT   | ALBERT   |
| TOMMY     | M      | 167    | ALBERT   | TOMMY    |
| BUDDY     | M      | 189    | ALBERT   | BUDDY    |
| FARQUAR   | M      | 198    | ALBERT   | FARQUAR  |
| SIMON     | M      | 256    | ALBERT   | SIMON    |
+-----------+--------+--------+----------+----------+
```

partition boundary

As you seen in the column `LT_CHILD`, `ROSEMARY` is returned as the lightest female and `ALBERT` is returned as the lightest male. But, something strange – like a funeral home offering curbside pickup – is happening with column `HV_CHILD`. The heaviest female is `LAUREN`, but `ROSEMARY`'s name seems to appear with `LAUREN`'s name under the column `HV_CHILD`. When looking at the boys under column `HV_CHILD`, you'll notice that the heaviest male child, `SIMON`, is mixed in with the names of the other boys. Something weird is going on!! This brings us to the *Windowing Clause*.

The Windowing Clause

You may have noticed that, in the examples above, the analytic functions either operated on data within the entire table or within a sliver of rows specified by the `PARTITION BY` Clause. The Windowing clause allows more fine-grained access to the rows of data being pushed into the analytic function. With the Windowing Clause, you aren't limited to an entire partition, but a portion of a partition. Now, despite its name, there's no `WINDOW` keyword. Instead, you'll make use of the `ROWS` and `RANGE` keywords.

Before we continue, let's go back to one of the first examples I showed you: running totals of `WEIGHT`:

```
SELECT A.GENDER,A.FIRSTNAME,A.WEIGHT,
       SUM(A.WEIGHT) OVER (PARTITION BY A.GENDER
                           ORDER BY A.WEIGHT) AS WT_RUN
 FROM FATKIDS A
 ORDER BY A.GENDER,A.WEIGHT;
```

```
+--------+-----------+--------+--------+
| gender | firstname | weight | wt_run |
+--------+-----------+--------+--------+
| F      | ROSEMARY  | 123    | 123    |
| F      | LAUREN    | 876    | 999    |
| M      | ALBERT    | 150    | 150    |
| M      | TOMMY     | 167    | 317    |
| M      | BUDDY     | 189    | 506    |
| M      | FARQUAR   | 198    | 704    |
| M      | SIMON     | 256    | 960    |
+--------+-----------+--------+--------+
```

partition boundary

This example may have thrown some of you for a loop because the inclusion of the `PARTITION BY` and `ORDER BY` Clauses doesn't really explain how the running totals are being computed. In fact, the `SUM` function, being used in an analytic sense above, is missing the Windowing Clause. Let me re-write the code above by adding in the **default** Windowing Clause:

```
SELECT A.GENDER,A.FIRSTNAME,A.WEIGHT,
       SUM(A.WEIGHT) OVER (PARTITION BY A.GENDER
                           ORDER BY A.WEIGHT
                           ROWS BETWEEN UNBOUNDED PRECEDING
                                    AND CURRENT ROW) AS WT_RUN
 FROM FATKIDS A
 ORDER BY A.GENDER,A.WEIGHT;
```

The default Windowing Clause indicates that the rows of data (`ROWS`) being forced into the analytic function is from the current row (`CURRENT ROW`) on back (`UNBOUNDED PRECEDING`). In the output above, let's say the current row is `BUDDY`'s row with a `WEIGHT` of `189`. The ***unbounded preceding*** is from `TOMMY`'s row on up based on the ordering of the data, here by ascending `WEIGHT`, to include `TOMMY`'s and `ALBERT`'s `WEIGHT` as well (since we're partitioning by `GENDER`, you must stay within the partition boundary). As the current row shifts down, the ***unbounded preceding*** contains more and more rows of data starting from the current row all the way up to the top (***unbounded preceding***). This is why the running total works because you're adding in more `WEIGHT`s as you move down the table. Remember: everything works within a **partition boundary** and data is reset as you cross the partition boundary.

So, let's get back to the `FIRST_VALUE` and `LAST_VALUE` example. Here's the code with the **default** Windowing Clause shoehorned in:

```
SELECT A.FIRSTNAME,A.GENDER,A.WEIGHT,
       FIRST_VALUE(A.FIRSTNAME) OVER (PARTITION BY A.GENDER
                                      ORDER BY A.WEIGHT
                                      ROWS BETWEEN UNBOUNDED PRECEDING
                                               AND CURRENT ROW) AS LT_CHILD,
       LAST_VALUE(A.FIRSTNAME) OVER (PARTITION BY A.GENDER
                                     ORDER BY A.WEIGHT
                                     ROWS BETWEEN UNBOUNDED PRECEDING
                                              AND CURRENT ROW) AS HV_CHILD
 FROM FATKIDS A
 ORDER BY A.GENDER,A.WEIGHT;
```

```
+-----------+--------+--------+----------+----------+
| firstname | gender | weight | lt_child | hv_child |
+-----------+--------+--------+----------+----------+
| ROSEMARY  | F      | 123    | ROSEMARY | ROSEMARY |
| LAUREN    | F      | 876    | ROSEMARY | LAUREN   |
| ALBERT    | M      | 150    | ALBERT   | ALBERT   |
| TOMMY     | M      | 167    | ALBERT   | TOMMY    |
| BUDDY     | M      | 189    | ALBERT   | BUDDY    |
| FARQUAR   | M      | 198    | ALBERT   | FARQUAR  |
| SIMON     | M      | 256    | ALBERT   | SIMON    |
+-----------+--------+--------+----------+----------+
```

partition boundary

Looking again at the `LT_CHILD` column, we see that the results are correct. The default Windowing Clause forces the rows from the **current row on back to the top** to be passed into the `FIRST_VALUE` analytic function. This is why `FIRST_VALUE` works. But, it's also why `LAST_VALUE` doesn't work and why the column `HV_CHILD` is screwed up *royal moose stylie*. Let's again say the current row is `BUDDY`'s row with a `WEIGHT` of `189`. The default Windowing Clause allows only the rows for `BUDDY`, `TOMMY` and `ALBERT` to be passed to the `LAST_VALUE` analytic function. What's the last value for those three rows? It's `BUDDY`'s row which is why his name appears in the `HV_CHILD` column. If you move down to `FARQUAR`'s row, what's the last value for `ALBERT`'s, `TOMMY`'s, `BUDDY`'s and

`FARQUAR`'s rows? It's `FARQUAR`'s row. And so on. This explains why the column `HV_CHILD` just contains the `FIRSTNAME` as you proceed down the partition.

So, how would you fix the issue with the `LAST_VALUE` analytic function? Well, you'd need to change the Windowing Clause so that the rows from the *current row on down to the end* are pushed into the `LAST_VALUE` analytic function. Here's how we fix that:

```
SELECT A.FIRSTNAME,A.GENDER,A.WEIGHT,
       FIRST_VALUE(A.FIRSTNAME) OVER (PARTITION BY A.GENDER
                                      ORDER BY A.WEIGHT
                                      ROWS BETWEEN UNBOUNDED PRECEDING
                                                AND CURRENT ROW) AS LT_CHILD,
       LAST_VALUE(A.FIRSTNAME) OVER (PARTITION BY A.GENDER
                                     ORDER BY A.WEIGHT
                                     ROWS BETWEEN CURRENT ROW
                                          AND UNBOUNDED FOLLOWING) AS HV_CHILD
 FROM FATKIDS A
 ORDER BY A.GENDER,A.WEIGHT;
```

```
+-----------+--------+--------+----------+----------+
| firstname | gender | weight | lt_child | hv_child |
+-----------+--------+--------+----------+----------+   partition
| ROSEMARY  | F      | 123    | ROSEMARY | LAUREN   |   boundary
| LAUREN    | F      | 876    | ROSEMARY | LAUREN   |
| ALBERT    | M      | 150    | ALBERT   | SIMON    |
| TOMMY     | M      | 167    | ALBERT   | SIMON    |
| BUDDY     | M      | 189    | ALBERT   | SIMON    |
| FARQUAR   | M      | 198    | ALBERT   | SIMON    |
| SIMON     | M      | 256    | ALBERT   | SIMON    |
+-----------+--------+--------+----------+----------+
```

The Windowing Clause for the `LAST_VALUE` analytic function was changed to `ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING`. And, as you can see, the column `HV_CHILD` is now correct. If you want to be completely anarchistic, you can specify `ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING`, which just passes the entire partition (or table) into the `LAST_VALUE` analytic function…not very fine-grained, but sometimes necessary.

Now, I mentioned that the Windowing Clause gives you more fine-grained access, but `ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` is less fine-grained and more sledgehammer. Here's the syntax for the `ROWS` Windowing Clause:

```
ROWS BETWEEN m PRECEDING | UNBOUNDED PRECEDING | CURRENT ROW
         AND CURRENT ROW | UNBOUNDED FOLLOWING | n FOLLOWING
```

The syntax *`m PRECEDING`* and *`n FOLLOWING`* indicate the number of rows off of the current row you want passed into the analytic function. For example, let's compute the average `WEIGHT` using the **current row**, **the prior row** and **the next row**:

```
SELECT A.FIRSTNAME,A.GENDER,A.WEIGHT,
       AVG(A.WEIGHT) OVER (PARTITION BY A.GENDER
                           ORDER BY A.WEIGHT
                           ROWS BETWEEN 1 PRECEDING
                                    AND 1 FOLLOWING) AS AVG_3
 FROM FATKIDS A
 ORDER BY A.GENDER,A.WEIGHT;
```

```
+-----------+--------+--------+-------------------+
| firstname | gender | weight | avg_3             |
+-----------+--------+--------+-------------------+
| ROSEMARY  | F      | 123    | 499.5             |
| LAUREN    | F      | 876    | 499.5             |
| ALBERT    | M      | 150    | 158.5             |
| TOMMY     | M      | 167    | 168.6666666666667 |
| BUDDY     | M      | 189    | 184.6666666666667 |
| FARQUAR   | M      | 198    | 214.3333333333333 |
| SIMON     | M      | 256    | 227               |
+-----------+--------+--------+-------------------+
```

partition boundary

Well…that was simple! Notice that fewer rows of data are passed into the `AVG` function depending on whether the partition boundary, top of table or bottom of table have been breached…like a modern day, security-focused website (`#sarcasm`). For `LAUREN`'s row, the average `WEIGHT` is computed from the values `123` and `876`. For `BUDDY`'s row, the average `WEIGHT` is computed from the values `189`, `167` and `198`.

An alternate Windowing Clause uses the `RANGE` keyword instead of `ROWS`. Whereas `ROWS` pushes a certain number of rows into an analytic function above and below the current row, the `RANGE` variation on the Windowing Clause pushes a certain number of rows into an analytic function based on the column(s) specified on `ORDER BY`. Now, unlike other legacy databases, the `RANGE` feature is very limited in ImpalaSQL at this time, and you're limited to using `CURRENT ROW`, `UNBOUNDED PRECEDING` and `UNBOUNDED FOLLOWING`.

For example, let's compute the average `WEIGHT` in the `FATKIDS` table based on the ordering of ascending `HEIGHT` using a `RANGE` Windowing Clause. First, let's see what the `ROWS` Windowing Clause will output:

```
SELECT A.FIRSTNAME,A.HEIGHT,A.WEIGHT,
       AVG(A.WEIGHT) OVER (ORDER BY A.HEIGHT
                           ROWS BETWEEN UNBOUNDED PRECEDING
                                    AND CURRENT ROW) AS SUM_10_5
 FROM FATKIDS A
 ORDER BY A.HEIGHT;

+-----------+--------+--------+-------------------+
| firstname | height | weight | sum_10_5          |
+-----------+--------+--------+-------------------+
| ROSEMARY  | 35     | 123    | 123               |
| ALBERT    | 45     | 150    | 136.5             |
| BUDDY     | 45     | 189    | 154               |
| LAUREN    | 54     | 876    | 334.5             |
| FARQUAR   | 76     | 198    | 307.2             |
| TOMMY     | 78     | 167    | 283.8333333333333 |
| SIMON     | 87     | 256    | 279.8571428571428 |
+-----------+--------+--------+-------------------+
```

For example, on `ALBERT`'s row, the average weight is `(123+150)/2 = 136.5`. And on `BUDDY`'s row, the average weight is `(123+150+189)/3 = 154`. Now, let's see `RANGE` in action:

```
SELECT A.FIRSTNAME,A.HEIGHT,A.WEIGHT,
       AVG(A.WEIGHT) OVER (ORDER BY A.HEIGHT
                           RANGE BETWEEN UNBOUNDED PRECEDING
                                     AND CURRENT ROW) AS SUM_10_5
 FROM FATKIDS A
 ORDER BY A.HEIGHT;

+-----------+--------+--------+-------------------+
| firstname | height | weight | sum_10_5          |
+-----------+--------+--------+-------------------+
| ROSEMARY  | 35     | 123    | 123               |
```

```
| ALBERT     | 45     | 150    | 154               |
| BUDDY      | 45     | 189    | 154               |
| LAUREN     | 54     | 876    | 334.5             |
| FARQUAR    | 76     | 198    | 307.2             |
| TOMMY      | 78     | 167    | 283.8333333333333 |
| SIMON      | 87     | 256    | 279.8571428571428 |
+-----------+--------+--------+-------------------+
```

If you take a look at BUDDY's row now, you'll see an average of 154, the same as for ROWS above. But, on ALBERT's row, we have 154 and not 136.5. This is because the RANGE Windowing Clause works with ORDER BY to determine the number of rows to pass into the AVG function. Since we're ordering by ascending HEIGHT, those values that are the same are passed into the AVG function as well. Since both ALBERT and BUDDY have a HEIGHT of 45, and we're specifying UNBOUNDED PRECEDING AND CURRENT ROW, the WEIGHTs 123, 150 and 189 are all passed into the AVG function to get an average of 154. This value is repeated on both ALBERT's and BUDDY's rows since they both have a HEIGHT of 45.

Statistics-Related Analytic Functions

There are several statistics-related functions which are used in an analytic sense: NTILE, PERCENT_RANK and CUME_DIST. Note that these functions require the ORDER BY Clause, the PARTITION BY Clause is optional and the Windowing Clause is not allowed.

The NTILE analytic function buckets the rows of your table with each bucket containing approximately CEIL(*nbr-rows*/*nbr-buckets*) number of rows. The buckets are numbered from one up to the specified number of *buckets*. The syntax for NTILE is:

```
NTILE(buckets) OVER ( … ORDER BY col1,col2, … )
```

The PARTITION BY Clause can be used, but is not required. The Windowing Clause is not allowed.

For example, let's use NTILE on the FATKIDS table to create four buckets based on ascending HEIGHT. Since CEIL(7/4)=2, we should have exactly two rows per bucket except for the last (seventh) row:

```
SELECT A.FIRSTNAME,A.HEIGHT,
       NTILE(4) OVER (ORDER BY A.HEIGHT) AS GRP4_HT
 FROM FATKIDS A
 ORDER BY A.HEIGHT;
```

```
+-----------+--------+---------+
| firstname | height | grp4_ht |
+-----------+--------+---------+
| ROSEMARY  | 35     | 1       |
| ALBERT    | 45     | 1       |
| BUDDY     | 45     | 2       |
| LAUREN    | 54     | 2       |
| FARQUAR   | 76     | 3       |
| TOMMY     | 78     | 3       |
| SIMON     | 87     | 4       |
+-----------+--------+---------+
```

Note that SIMON is the only child who appears in the fourth bucket (because he constantly *says* this and *says* that).

The PERCENT_RANK analytic function bases its computation on the RANK function to return a column containing values from zero to one. The formula used to compute the PERCENT_RANK is (*rank-nbr* − 1) / (*nbr-rows* − 1). Despite the name, a percentage is not returned, but a proportion. Here is the syntax:

```
PERCENT_RANK() OVER ( … ORDER BY col1,col2,… )
```

If you want a percentage, then use the following syntax:

```
100*PERCENT_RANK() OVER ( … ORDER BY col1,col2,… )
```

The PARTITION BY Clause can be used, but is not required. The Windowing Clause is not allowed.

For example, let's compute the PERCENT_RANK based on ascending HEIGHT:

```
SELECT A.FIRSTNAME,A.HEIGHT,
       RANK() OVER (ORDER BY A.HEIGHT) AS RANK_HEIGHT,
       PERCENT_RANK() OVER (ORDER BY A.HEIGHT) AS PCTDIST_HEIGHT
 FROM FATKIDS A
 ORDER BY A.HEIGHT;
```

```
+-----------+--------+-------------+--------------------+
| firstname | height | rank_height | pctdist_height     |
+-----------+--------+-------------+--------------------+
| ROSEMARY  | 35     | 1           | 0                  |
| BUDDY     | 45     | 2           | 0.1666666666666667 |
| ALBERT    | 45     | 2           | 0.1666666666666667 |
| LAUREN    | 54     | 4           | 0.5                |
| FARQUAR   | 76     | 5           | 0.6666666666666666 |
| TOMMY     | 78     | 6           | 0.8333333333333334 |
| SIMON     | 87     | 7           | 1                  |
+-----------+--------+-------------+--------------------+
```

To show you the calculations, the RANK function's output is displayed as well. Since there are seven rows in the FATKIDS table, the first row's PERCENT_RANK is calculated as (1 - 1)/(7 - 1) = 0. The row for LAUREN is calculated as (4 - 1)/(7 - 1) = 3/6 = 0.5. Note that those rows with the same RANK will have the same PERCENT_RANK value (as you can see for BUDDY's and ALBERT's rows).

The CUME_DIST analytic function is computed based on the number of rows that are less than or equal to the column you're providing divided by the total number of rows. The approximate formula is *row-nbr/nbr-rows* and the resulting values range from just above zero to exactly one. The syntax is:

```
CUME_DIST(column) OVER ( … ORDER BY col1,col2,… )
```

The PARTITION BY Clause can be used, but is not required. The Windowing Clause is not allowed.

For example, let's compute the cumulative distribution based on the HEIGHT:

```
SELECT A.FIRSTNAME,A.HEIGHT,
       CUME_DIST() OVER (ORDER BY A.HEIGHT) AS CUMDIST_HEIGHT
 FROM FATKIDS A
 ORDER BY A.HEIGHT;
```

```
+-----------+--------+--------------------+
| firstname | height | cumdist_height     |
+-----------+--------+--------------------+
| ROSEMARY  | 35     | 0.1428571428571428 |
| ALBERT    | 45     | 0.4285714285714285 |
| BUDDY     | 45     | 0.4285714285714285 |
| LAUREN    | 54     | 0.5714285714285714 |
| FARQUAR   | 76     | 0.7142857142857143 |
| TOMMY     | 78     | 0.8571428571428571 |
| SIMON     | 87     | 1                  |
+-----------+--------+--------------------+
```

For LAUREN's row, there are 4 rows less than or equal to her height of 54 divided by 7 total rows: 4/7 = .5714. Note that both ALBERT and BUDDY have the same HEIGHT value of 45 and each receives the maximum CUME_DIST based off of BUDDY's row: 3/7 = .4285.

The Elusive QUALIFY Clause

Recall that both the WHERE and HAVING Clauses allow you to subset data. When creating a column based on the analytic function techniques shown in this chapter, the ability to subset data based on its results usually requires you to surround the entire query with a SELECT * FROM (…*query*…) A and then tack on a subsetting WHERE Clause based on the new column(s). The QUALIFY Clause allows you to bag the surrounding query and subset the data based on one or more columns created using analytic function techniques. Unfortunately, the QUALIFY Clause is, at the time of publication, only available in the Teradata, BigQuery, Snowflake and Databricks databases. You may want to occasionally check the HiveQL and ImpalaSQL online documentation to see if the QUALIFY Clause has become available in your database.

For example, recall our first query using an analytic function:

```
SELECT FIRSTNAME,GENDER,BIRTHDATE,HEIGHT,WEIGHT,
       COUNT(*) OVER (PARTITION BY GENDER) AS KID_COUNT
 FROM FATKIDS;
```

In order to subset based on the column KID_COUNT, one approach is the following:

```
SELECT A.*
 FROM (
       SELECT FIRSTNAME,GENDER,BIRTHDATE,HEIGHT,WEIGHT,
              COUNT(*) OVER (PARTITION BY GENDER) AS KID_COUNT
        FROM FATKIDS
      ) A
 WHERE KID_COUNT=2;
```

But, if the QUALIFY Clause is available, you can get away with the following:

```
SELECT FIRSTNAME,GENDER,BIRTHDATE,HEIGHT,WEIGHT,
       COUNT(*) OVER (PARTITION BY GENDER) AS KID_COUNT
 FROM FATKIDS
 QUALIFY KID_COUNT=2;
```

You can also do the following:

```
SELECT FIRSTNAME,GENDER,BIRTHDATE,HEIGHT,WEIGHT,
       COUNT(*) OVER (PARTITION BY GENDER) AS KID_COUNT
 FROM FATKIDS
 QUALIFY COUNT(*) OVER (PARTITION BY GENDER)=2;
```

Sweet!!

Chapter 13 – Extensions to the `GROUP BY` Clause in ImpalaSQL

As indicated in the Hadoop Administrator e-mail, your Hadoop database's installed version of ImpalaSQL may or may not have the extensions to the `GROUP BY` Clause available. If not, you may want to skip this chapter upon first read. Be aware that the extensions to the `GROUP BY` Clause are available in HiveQL.

As you're well aware, the `GROUP BY` Clause returns rows of data based solely on the indicated columns. If you need additional summarizations, you can use the fab `UNION ALL` Clause to append those rows. Depending on the number of summarizations you need, your SQL query can become very large. And, as we all know, cutting and pasting is fraught with problems and a mistake is bound to happen.

To mitigate this risk, and prevent war from breaking out in Toronto, there are extensions to the `GROUP BY` Clause which allow you to succintly specify all of the summarizations you need…without all of those damn `UNION ALL`s! The extentions are `GROUP BY` **`CUBE`**, `GROUP BY` **`ROLLUP`** and `GROUP BY` **`GROUPING SETS`**. We discuss each extension in turn below.

Data Used in this Chapter

In this chapter, we use the completely fake, yet scrumptious, Candybar Consumption dataset, described below. The fake table name is `prod_schema.candybar_consumption_data` and has the following fake columns:

- `CONSUMER_ID` – A unique identifier or each candybar consumer.
- `CANDYBAR_NAME` – The name of the candy bar consumed (e.g., `MARS BAR`, `TWIX BAR`, …).
- `SURVEY_YEAR` – The year of survey responses (e.g., `2009`, `2010`, …).
- `GENDER` – The gender of respondent (e.g., `M`=Male, `F`=Female).
- `OVERALL_RATING` – The rating of the candybar ranging from `1`=Low to `10`=High.
- `NUMBER_BARS_CONSUMED` – The number of candybars consumed for the year.

```
[hdpserver.com:21000] prod_schema> desc candybar_consumption_data;
+----------------------+----------+---------+
| name                 | type     | comment |
+----------------------+----------+---------+
| consumer_id          | tinyint  |         |
| candybar_name        | string   |         |
| survey_year          | smallint |         |
| gender               | string   |         |
| overall_rating       | tinyint  |         |
| number_bars_consumed | smallint |         |
+----------------------+----------+---------+

+-------------+---------------+-------------+--------+----------------+----------------------+
| consumer_id | candybar_name | survey_year | gender | overall_rating | number_bars_consumed |
+-------------+---------------+-------------+--------+----------------+----------------------+
| 1           | MARS BAR      | 2009        | M      | 10             | 252                  |
| 1           | MARS BAR      | 2010        | M      | 10             | 352                  |
| 1           | MARS BAR      | 2011        | M      | 10             | 452                  |
| 1           | TWIX BAR      | 2009        | M      | 10             | 6                    |
| 1           | TWIX BAR      | 2010        | M      | 7              | 60                   |
| 1           | TWIX BAR      | 2011        | M      | 8              | 600                  |
| 2           | HERSHEY BAR   | 2009        | F      | 5              | 2                    |
| 2           | HERSHEY BAR   | 2010        | F      | 5              | 3                    |
| 2           | HERSHEY BAR   | 2011        | F      | 5              | 1                    |
| 2           | MARS BAR      | 2009        | F      | 8              | 25                   |
| 2           | MARS BAR      | 2010        | F      | 8              | 12                   |
| 2           | MARS BAR      | 2011        | F      | 8              | 13                   |
| 3           | MARS BAR      | 2009        | M      | 8              | 25                   |
| 3           | MARS BAR      | 2010        | M      | 7              | 12                   |
| 3           | MARS BAR      | 2011        | M      | 8              | 13                   |
| 3           | TWIX BAR      | 2009        | M      | 7              | 6                    |
| 3           | TWIX BAR      | 2010        | M      | 8              | 60                   |
| 3           | TWIX BAR      | 2011        | M      | 9              | 600                  |
| 4           | HERSHEY BAR   | 2009        | F      | 7              | 20                   |
| 4           | HERSHEY BAR   | 2010        | F      | 7              | 30                   |
| 4           | HERSHEY BAR   | 2011        | F      | 7              | 10                   |
```

```
| 4           | MARS BAR      | 2009       | F      | 7             | 25                  |
| 4           | MARS BAR      | 2010       | F      | 7             | 35                  |
| 4           | MARS BAR      | 2011       | F      | 7             | 15                  |
| 4           | TWIX BAR      | 2009       | F      | 7             | 20                  |
| 4           | TWIX BAR      | 2010       | F      | 7             | 30                  |
| 4           | TWIX BAR      | 2011       | F      | 7             | 10                  |
| 5           | HERSHEY BAR   | 2009       | M      | 8             | 15                  |
| 5           | HERSHEY BAR   | 2010       | M      | 8             | 15                  |
| 5           | HERSHEY BAR   | 2011       | M      | 6             | 5                   |
| 5           | SNICKERS BAR  | 2009       | M      | 8             | 55                  |
| 5           | SNICKERS BAR  | 2010       | M      | 8             | 65                  |
| 5           | SNICKERS BAR  | 2011       | M      | 8             | 75                  |
| 5           | TWIX BAR      | 2009       | M      | 9             | 75                  |
| 5           | TWIX BAR      | 2010       | M      | 9             | 85                  |
| 5           | TWIX BAR      | 2011       | M      | 9             | 95                  |
+-------------+---------------+------------+--------+---------------+---------------------+
```

Note: The Surgeon General, as well as Mom General, recommends not eating 600 candybars per year.

Motivational Example

Let's use the vintage GROUP BY to sum the NUMBER_BARS_CONSUMED down to the SURVEY_YEAR, CANDYBAR_NAME, GENDER and OVERALL_RATING level:

```
SELECT SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING,
       SUM(NUMBER_BARS_CONSUMED) AS TOTAL_BARS_CONSUMED
 FROM CANDYBAR_CONSUMPTION_DATA
 GROUP BY SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING
 ORDER BY 1,2,3,4;

+-------------+---------------+--------+----------------+---------------------+
| survey_year | candybar_name | gender | overall_rating | total_bars_consumed |
+-------------+---------------+--------+----------------+---------------------+
| 2009        | HERSHEY BAR   | F      | 5              | 2                   |
| 2009        | HERSHEY BAR   | F      | 7              | 20                  |
| 2009        | HERSHEY BAR   | M      | 8              | 15                  |
| 2009        | MARS BAR      | F      | 7              | 25                  |
| 2009        | MARS BAR      | F      | 8              | 25                  |
...snip...
| 2011        | TWIX BAR      | F      | 7              | 10                  |
| 2011        | TWIX BAR      | M      | 8              | 600                 |
| 2011        | TWIX BAR      | M      | 9              | 695                 |
+-------------+---------------+--------+----------------+---------------------+
```

As you see, the data has been summarized to the SURVEY_YEAR, CANDYBAR_NAME, GENDER and OVERALL_RATING level and only that level. Next, let's add in some summary levels as well as a grand total:

```
SELECT SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING,
       SUM(NUMBER_BARS_CONSUMED) AS TOTAL_BARS_CONSUMED
 FROM CANDYBAR_CONSUMPTION_DATA
 GROUP BY SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING
UNION ALL
SELECT SURVEY_YEAR,CANDYBAR_NAME,GENDER,NULL AS OVERALL_RATING,
       SUM(NUMBER_BARS_CONSUMED) AS TOTAL_BARS_CONSUMED
 FROM CANDYBAR_CONSUMPTION_DATA
 GROUP BY SURVEY_YEAR,CANDYBAR_NAME,GENDER
UNION ALL
SELECT SURVEY_YEAR,CANDYBAR_NAME,NULL AS GENDER,NULL AS OVERALL_RATING,
       SUM(NUMBER_BARS_CONSUMED) AS TOTAL_BARS_CONSUMED
 FROM CANDYBAR_CONSUMPTION_DATA
 GROUP BY SURVEY_YEAR,CANDYBAR_NAME
```

```
UNION ALL
SELECT SURVEY_YEAR,NULL AS CANDYBAR_NAME,NULL AS GENDER,NULL AS
       OVERALL_RATING, SUM(NUMBER_BARS_CONSUMED) AS TOTAL_BARS_CONSUMED
 FROM CANDYBAR_CONSUMPTION_DATA
 GROUP BY SURVEY_YEAR
UNION ALL
SELECT NULL AS SURVEY_YEAR,NULL AS CANDYBAR_NAME,NULL AS GENDER,NULL AS
       OVERALL_RATING,SUM(NUMBER_BARS_CONSUMED) AS TOTAL_BARS_CONSUMED
 FROM CANDYBAR_CONSUMPTION_DATA;
```

You'll notice that we've made use of UNION ALL to append additional summary rows (the second, third, fourth and fifth SQL queries in the code above) to the original SQL query. In order for UNION ALL to work, all of the columns need to appear which is why I'm making judicious use of NULL. Each query, from the first to the fifth, summarizes the data to a higher and higher level starting from all of the columns (SURVEY_YEAR, CANDYBAR_NAME, GENDER and OVERALL_RATING) up to the grand total (all columns are NULL). A portion of the results appears below:

```
+-------------+---------------+--------+----------------+---------------------+
| survey_year | candybar_name | gender | overall_rating | total_bars_consumed |
+-------------+---------------+--------+----------------+---------------------+
| 2009        | HERSHEY BAR   | F      | 5              | 2                   |
...snip...
| 2010        | TWIX BAR      | M      | NULL           | 205                 |
...snip...
| 2009        | MARS BAR      | NULL   | NULL           | 327                 |
...snip...
| 2010        | NULL          | NULL   | NULL           | 759                 |
...snip...
| NULL        | NULL          | NULL   | NULL           | 3174                |
+-------------+---------------+--------+----------------+---------------------+
```

Now, let's recreate the same output using the ROLLUP extension to the GROUP BY Clause:

```
SELECT SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING,
       SUM(NUMBER_BARS_CONSUMED) AS TOTAL_BARS_CONSUMED
 FROM CANDYBAR_CONSUMPTION_DATA
 GROUP BY ROLLUP(SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING);
```

Well, that was easy! We explain ROLLUP in more detail below, but as you can see, we included the ROLLUP keyword along with the relevant columns in parentheses.

GROUPING SETS

As far as the entensions to GROUP BY go, GROUPING SETS is the most generic allowing you to specify exactly the summaries you want without any unneeded summaries. This is in contrast to both ROLLUP and CUBE which may produce more summary data than you really want.

The syntax for GROUPING SETS is as follows:

```
GROUP BY GROUPING SETS(A,B,C,…)
```

where A, B, C, etc. can be either a single column, or a parenthesized list of columns separated by commas. The syntax above is equivalent to the following:

```
GROUP BY A
UNION ALL
GROUP BY B
UNION ALL
GROUP BY C
```

…and so on…

Note that GROUPING SETS does not produce the grand total, unlike both ROLLUP and CUBE. If you want a grand total, you must include an empty set of parentheses. For example, to reproduce the motivational example above, we can use GROUPING SETS:

```
SELECT SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING,
       SUM(NUMBER_BARS_CONSUMED) AS TOTAL_BARS_CONSUMED
 FROM CANDYBAR_CONSUMPTION_DATA
 GROUP BY GROUPING SETS(
                        (SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING),
                        (SURVEY_YEAR,CANDYBAR_NAME,GENDER),
                        (SURVEY_YEAR,CANDYBAR_NAME),
                        (SURVEY_YEAR),
                        () <------------ WHOA!! GRAND TOTAL!!
                       )
```

Take note that GROUPING SETS starts and ends with its own set of parentheses. Within these parentheses is one or more parenthesized lists of columns delimited by commas. Each parenthesized list is equivalent to the corresponding GROUP BY Clause along with a UNION ALL. The final parenthesized list shown above is empty, **()**, indicating that a grand total is desired.

Now, the code shown above is equivalent to the ROLLUP extension, so it'd be silly to use GROUPING SETS in this case. But, remember, with GROUPING SETS you can specify exactly the summaries you want and *t' heck with the rest*, for example:

```
SELECT SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING,
       SUM(NUMBER_BARS_CONSUMED) AS TOTAL_BARS_CONSUMED
 FROM CANDYBAR_CONSUMPTION_DATA
 GROUP BY GROUPING SETS(
                        (SURVEY_YEAR,CANDYBAR_NAME,GENDER),
                        (SURVEY_YEAR),
                        () <------------ WHOA!! GRAND TOTAL!!
                       )
```

ROLLUP

The ROLLUP extension to the GROUP BY Clause produces summaries useful for rollup reports. The syntax for ROLLUP is as follows:

```
GROUP BY ROLLUP(A,B,C,…)
```

and is equivalent to the following:

```
GROUPING SETS(
              (), <------------ WHOA!! GRAND TOTAL!!
              (A),
              (A,B),
              (A,B,C),
              …
             )
```

Take note that the grand total is automatically provided when you use ROLLUP.

Now, if this is confusing, you can visualize ROLLUP like this:

```
GROUP BY ROLLUP(A,B,C)
```

where the first arrow indicates `GROUP BY A,B,C`; the next arrow indicates `GROUP BY A,B`; the next arrow indicates `GROUP BY A`; and, finally, the castrated arrow indicates the grand total. You can extend this concept – as well as the arrows – with more than three columns.

Let's recreate the exact output from the motivational example:

```
SELECT SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING,
       SUM(NUMBER_BARS_CONSUMED) AS TOTAL_BARS_CONSUMED
 FROM CANDYBAR_CONSUMPTION_DATA
 GROUP BY ROLLUP(SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING)
```

Note that you may have produced rollup reports by using the pivot table feature in Microsoft Excel. Now, you can do a similar thing using SQL without the `UNION ALL`s. Nice!

CUBE

The `CUBE` extension to the `GROUP BY` Clause produces summaries based on the provided columns one at a time, two at a time, and so on; that is, **all combinations of the columns are produced**. The syntax for `CUBE` is as follows:

```
GROUP BY CUBE(A,B,C,…)
```

and is equivalent to the following:

```
GROUPING SETS(
              (),  <------------ WHOA!! GRAND TOTAL!!
              (A),(B),(C),…
              (A,B),(A,C),(B,C),…
              (A,B,C),…
              …
             )
```

Take note that the grand total is automatically produced when you use `CUBE`.

For example, let's produce summaries based all combinations of `SURVEY_YEAR`, `CANDYBAR_NAME`, `GENDER` and `OVERALL_RATING`:

```
SELECT SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING,
       SUM(NUMBER_BARS_CONSUMED) AS TOTAL_BARS_CONSUMED
 FROM CANDYBAR_CONSUMPTION_DATA
 GROUP BY CUBE(SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING)
```

From this single SQL query, you'll produce the following summaries:

- ☐ 1-at-a-time summaries: `SURVEY_YEAR`, `CANDYBAR_NAME`, `GENDER` and `OVERALL_RATING`
- ☐ 2-at-a-time summaries: `SURVEY_YEAR/CANDYBAR_NAME`, `SURVEY_YEAR/GENDER`, `SURVEY_YEAR/OVERALL_RATING`, `CANDYBAR_NAME/GENDER`, `CANDYBAR_NAME/OVERALL_RATING`

- 3-at-a-time summaries: SURVEY_YEAR/CANDYBAR_NAME/GENDER, SURVEY_YEAR/CANDYBAR_NAME/OVERALL_RATING, CANDYBAR_NAME/GENDER/OVERALL_RATING
- 4-at-a-time summaries: SURVEY_YEAR/CANDYBAR_NAME/GENDER/OVERALL_RATING
- Grand total

GROUPING() and GROUPING_ID() Functions

You'll notice that the output from GROUPING SETS, ROLLUP and CUBE all use NULL values to indicate the summary rows. This is great if your data doesn't contain any NULLs, but this is the real world, pal, and NULLs are everywhere!

In order to determine which rows are actually the summary rows, ImpalaSQL provides two functions:

- GROUPING(*column-name*) – This function takes the name of a column as its sole argument and returns 1 if the row is a summary row for that column; otherwise, 0 is returned indicating the row is not a summary row for that column. A minor tragedy of this function is that you need to create additional columns, one for each column used in GROUPING SETS, CUBE or ROLLUP to get a complete picture of the summaryness. This is where the GROUPING_ID() function beams in to save the day (described below). For example, let's add four GROUPING() columns to our ROLLUP example:

```
SELECT SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING,
       SUM(NUMBER_BARS_CONSUMED) AS TBC,
       GROUPING(SURVEY_YEAR) AS gS,
       GROUPING(CANDYBAR_NAME) AS gC,
       GROUPING(GENDER) AS gG,
       GROUPING(OVERALL_RATING) AS gO
 FROM CANDYBAR_CONSUMPTION_DATA
 GROUP BY ROLLUP(SURVEY_YEAR,CANDYBAR_NAME,GENDER,OVERALL_RATING);
```

A subset of the results appears below with the additional four GROUPING() columns:

```
+-------------+---------------+--------+----------------+--------+------+------+------+------+
| survey_year | candybar_name | gender | overall_rating | tbc    | gS   | gC   | gG   | gO   |
+-------------+---------------+--------+----------------+--------+------+------+------+------+
...snip...
| 2009        | HERSHEY BAR   | F      | 5              | 2      | 0    | 0    | 0    | 0    |
| 2010        | TWIX BAR      | M      | NULL           | 205    | 0    | 0    | 0    | 1    |
| 2009        | MARS BAR      | NULL   | NULL           | 327    | 0    | 0    | 1    | 1    |
| 2010        | NULL          | NULL   | NULL           | 759    | 0    | 1    | 1    | 1    |
| NULL        | NULL          | NULL   | NULL           | 3174   | 1    | 1    | 1    | 1    |
+-------------+---------------+--------+----------------+--------+------+------+------+------+
```

Whenever you see a 1 in the GROUPING() columns, it's an indication that that particular row is a summary row. Now, the level of summarization depends on how many 1s there are. For example, when gG=1 and gO=1, that row is summarized down to the SURVEY_YEAR and CANDYBAR_NAME level (these two columns have zero for their GROUPING() values indicating these two columns were used to produce the summary, but the other two columns were not used).

Now, let's just focus on the four GROUPING() columns gS, gC, gG and gO for a moment. Let's slam these four columns together, like this:

```
+------+------+------+------+
|  gS  |  gC  |  gG  |  gO  |
+------+------+------+------+
|  0   |  0   |  0   |  0   |  → 0000
|  0   |  0   |  0   |  1   |  → 0001
|  0   |  0   |  1   |  1   |  → 0011
|  0   |  1   |  1   |  1   |  → 0111
|  1   |  1   |  1   |  1   |  → 1111
+------+------+------+------+
```

The emboldened column to the right may look suspiciously binary to you. If we pretend that they are actually binary numbers, let's convert them to decimal numbers and see what we get:

```
+------+------+------+------+
|  gS  |  gC  |  gG  |  gO  |
+------+------+------+------+
|  0   |  0   |  0   |  0   | ➔ 0000 ➔  0
|  0   |  0   |  0   |  1   | ➔ 0001 ➔  1
|  0   |  0   |  1   |  1   | ➔ 0011 ➔  3
|  0   |  1   |  1   |  1   | ➔ 0111 ➔  7
|  1   |  1   |  1   |  1   | ➔ 1111 ➔ 15
+------+------+------+------+
```

The decimal values shown above are exactly what the `GROUPING_ID()` function returns (described below).

- ☐ `GROUPING_ID(col1,col2,…)` – This function takes the list of columns used with `GROUPING SETS`, `ROLLUP` or `CUBE` as their argument and returns a single value indicating the columns used to produce each summary row. The value returned by this function is easier to work with when subsetting using a `WHERE` Clause is desired. In the example above, the value `15` is represented by the binary value `1111` and indicates that that particular row is a grand total row.

Chapter 14 – The One About HiveQL

Up to now, we've been discussing ImpalaSQL exclusively. Recall we mentioned that HiveQL is a bit of a sloth compared to ImpalaSQL and, while we'll mostly avoid HiveQL because of this, there are instances where you'll need it, such as importing data using regular expresssions or data in JSON format. (See *Chapter 23 – Working with Managed and External Tables* for more.) Also, the `LOAD DATA` Statement in HiveQL accepts the `LOCAL` keyword allowing you to load data directly from the Linux filesystem, a feature not available from ImpalaSQL. (See *Chapter 30 – Loading Data using LOAD DATA to Load Data* for more.)

Although we won't perform the same bloodsoaked autopsy on HiveQL as we did for ImpalaSQL across the previous few chapters, we do discuss a few things about HiveQL you should know. Please see the HiveQL documentation for a very lengthy coroner's report.

Logging in Using `beeline`

In a similar vein to Impala, you can access Hive via a SQL Client GUI, the Hadoop User Experience (Hue) web interface or the Linux command line utility `beeline`. In this section, we show you how to access Hive to issue HiveQL commands using the Linux command line utility `beeline`. Be aware that the old command `hive` is now deprecated in favor of the command `beeline`.

After logging into the Linux edge node server via PuTTY, depending on how your genuine-OEM Hadoop Admininstrator has set things up, you may be able to simply type in `beeline` at the Linux command line to access Hive to issue HiveQL queries:

```
[smithbob@lnxserver ~]$ beeline
beeline>
```

Sadly, you may be shown the `beeline` command prompt above, but not actually be connected to Hive. Issuing `show tables;` may return the following message:

```
beeline> show tables;
No current connection
```

To quit out of `beeline`, enter **`!q`** and hit the Enter key.

A better way to connect is by providing your username, password, schema and the Hive host and port, like this:

```
[smithbob@lnxserver ~]$ beeline –u jdbc:hive2://hdpserver:10000/schema
                                                    username password
```

Alternatively, you can specify your username and password using the `-n` and `-p` switches:

```
[smithbob@lnxserver ~]$ beeline –u jdbc:hive2://hdpserver:10000/schema
                                 -n username
                                 -p password
```

If your company uses the Kerberos computer-network authentication protocol, you need to specify the `principle` option as part of the JDBC connection to Hive. Please see the responses to the Hadoop Administrator E-Mail in *Chapter 2 – Hadoop Administrator E-Mail* for more on Kerberos.

```
beeline –u jdbc:hive2://hdpserver:10000/schema;principle=...
        -n username
        -p password
```

Please work with your supermodelly Hadoop Administrator if you're having problems logging in.

Data Manipulation Language (DML)

SQL DML available in HiveQL is similar to that of ImpalaSQL with a few differences. In HiveQL, the SQL Data Manipulation Language (DML) syntax looks broadly like the following:

```
WITH Clause
SELECT col1,col2,...
 FROM tbl_name
 WHERE subsetting_conditions
 GROUP BY col1,... [ CUBE() | ROLLUP() | GROUPING SETS() ]
 HAVING post_subsetting_conditions
 ORDER BY col1,...
 LIMIT offset,rows
 TABLESAMPLE(# ROWS)
```

At this point, most of these statements should be apparent, but take note of the `LIMIT` and `TABLESAMPLE` Clauses.

For the `LIMIT` Clause, you can specify both a *row offset* as well as the *number of desired rows*. If you specify just one value, offset defaults to zero, and the number of desired rows is returned:

```
beeline> select category_id from categories limit 10;
+--------------+
| category_id  |
+--------------+
| 1            |
| 2            |
| 3            |
| 4            |
| 5            |
| 6            |
| 7            |
| 8            |
| 9            |
| 10           |
+--------------+
```

If you specify both values, the first is the offset row within the table followed by the number of desired rows:

```
beeline> select category_id from categories limit 1,10;
+--------------+
| category_id  |                                    offset  rows
+--------------+
| 2            |
| 3            |
| 4            |
| 5            |
| 6            |
| 7            |
| 8            |
| 9            |
| 10           |
| 11           |
+--------------+
```

Don't forget that you can use the `ORDER BY` Clause with the `LIMIT` Clause to return rows in a desired order.

The `TABLESAMPLE` Clause allows you to request a random number of rows, which differs from ImpalaSQL's `TABLESAMPLE` Clause syntax. Note that HiveQL allows for syntax to select by input pruning as well as block

sampling. Please see the HiveQL documentation for more. To select a random number of rows, you specify the `TABLESAMPLE` Clause along with a desired number of rows followed by the `ROWS` keyword:

```
beeline > select category_id from categories tablesample(10 rows);
+--------------+
| category_id  |
+--------------+
| 1            |
| 2            |
| 3            |
| 4            |
| 5            |
| 6            |
| 7            |
| 8            |
| 9            |
| 10           |
+--------------+
```

Note that the `TABLESAMPLE` Clause must follow the `FROM` Clause. As you can see, the output above isn't very rAn𝒟O*m*, but you can use the `LIMIT` Clause after an `ORDER BY` Clause providing the `rand()` function to achieve something similar, yet random:

```
beeline> select category_id from categories order by rand() limit 10;
+--------------+
| category_id  |
+--------------+
| 13           |
| 29           |
| 10           |
| 52           |
| 19           |
| 9            |
| 41           |
| 6            |
| 48           |
| 36           |
+--------------+
```

Ah! This week's winning lottery numbers!!

Data Definition Language (DDL)

The DDL available in HiveQL is similar to that of ImpalaSQL. As in the section for ImpalaSQL, let's start off discussing the data types in HiveQL. Now, the data types available in HiveQL are similar to those of ImpalaSQL with a few minor exceptions, as shown below:

HiveQL Data Types

- ☐ **Integral Data Types**
  - `INTEGER` – This is an alias for `INT`.
- ☐ **Floating Point Data Types**
  - `DOUBLE PRECISION` – This is an alias for `DOUBLE`.
- ☐ **Decimal Data Type**
  - `NUMERIC` – This is an alias for `DECIMAL`.

Note that the HiveQL documentation intimates that INTERVAL is a fully-fledged data type just like in Oracle, but this is incorrect. The INTERVAL keyword acts in a similar manner as in ImpalaSQL functions. With that said, you may find, occasionally, that when describing a table, the following strange data types may appear: interval_day_time and interval_year_month. This may occur when using the CTAS syntax to create a table and one or more of your columns makes use of the INTERVAL keyword. You cannot use interval_day_time and interval_year_month as data types as yet, so I would avoid this particular *feature* for now.

HiveQL Functions

HiveQL offers many additional functions over those available in ImpalaSQL. As far as the aggregate functions, our familiar friends COUNT, AVG, MIN, MAX, SUM, etc. are available. You may want to peruse the HiveQL list of functions just in case something catches your eye and makes you shudder with excitement.

Additional statistical functions are available, such as the CORR function which allows you to compute the Pearson correlation between two columns. The REGR_* functions allow you to compute a simple linear regression between a single dependent and independent variable and retrieve the slope, intercept, R^2, and more: REGR_SLOPE, REGR_INTERCEPT, REGR_R2, etc.

For example, let's compute the correlation between the average number of candybars consumed for the males and the females across SURVEY_YEAR and CANDYBAR_NAME:

```
with vwBOYS as (
                select survey_year,candybar_name,
                                     avg(number_bars_consumed) as avgbars_boys
                 from candybar_consumption_data
                 where gender='M'
                 group by survey_year,candybar_name
               ),
     vwGALS as (
                select survey_year,candybar_name,
                                     avg(number_bars_consumed) as avgbars_gals
                 from candybar_consumption_data
                 where gender='F'
                 group by survey_year,candybar_name
               ),
     vwDATA as (
               select A.survey_year,A.candybar_name,
                      A.avgbars_boys,B.avgbars_gals
                from vwBOYS A inner join vwGALS B
                on A.survey_year=B.survey_year
                   and A.candybar_name=B.candybar_name
               )
select corr(avgbars_boys,avgbars_gals) as corr_avgbars_boys_gals
 from vwDATA;

+-------------------------+
| corr_avgbars_boys_gals  |
+-------------------------+
| -0.10161457962249516    |
+-------------------------+
```

You'll all be happy to know that the value above matches Microsoft Excel! Woo-hoo!!

Storage Formats

Similar to ImpalaSQL, you can use the `STORED AS` Clause to specify a built-in storage format: `TEXTFILE`, `PARQUET`, `AVRO`, `JSONFILE`, `ORC`, `RCFILE`, and `SEQUENCEFILE`. Note that `KUDU` is not available, but can be access by specifying the correct *serde*. Please work with your cranially-enhanced Hadoop Administrator if you plan on using the `KUDU` storage format from HiveQL. If a desired storage format is not available, you can specify the appropriate Java classes for the *input format*, *output format* and *serde* directly. Please see *Chapter 23 – Working with Managed and External Tables* for detailed information.

Creating Primary Keys and Indexes

Unlike ImpalaSQL, HiveQL allows you to create primary keys and indexes on one or more columns of a table. To create a primary key, specify the `PRIMARY KEY` Clause after the last column defined for the table. For example, let's recreate the `CANDYBAR_CONSUMPTION_DATA` table specifying `CONSUMER_ID`, `CANDYBAR_NAME` and `SURVEY_YEAR` as the primary keys:

```
create table candybar_consumption_data_new(consumer_id tinyint,
                                           candybar_name string,
                                           survey_year smallint,
                                           gender string,
                                           overall_rating tinyint,
                                           number_bars_consumed smallint,
        primary key(consumer_id,candybar_name,survey_year) disable novalidate)
 stored as parquet;
```

COMMA!!

The keywords `DISABLE` and `NOVALIDATE` are required. If you describe the table formatted, you'll see the primary key indicated near the bottom of the output:

```
| # Primary Key              | NULL                                            | NULL                        |
| Table:                     | prod_schema.candybar_consumption_data_new       | NULL                        |
| Constraint Name:           | pk_1809580168_1654543315847_0                   | NULL                        |
| Column Names:              | consumer_id                                     | candybar_name               |
+----------------------------+-------------------------------------------------+-----------------------------+
```

Take note of the primary key constraint name in the output above: `pk_1809580168_1654543315847_0`. (Memorize this as there may be a quiz later.)

If you're using Hive version 2, to create a simple or composite index on a table, use the `CREATE INDEX` Statement. At its simplest, the syntax looks like this:

```
CREATE INDEX index_name
 ON TABLE table_name (col1,col2,...)
 AS index_type
 WITH DEFERRED REBUILD
```

The `AS index_type` Clause allows you to specify whether the index is a value/block_id index (`'COMPACT'`) or a value/row_id index (`'BITMAP'`). Recall in other SQL flavors, to create a bitmap index you'd code the `CREATE BITMAP INDEX` Statement and to create a B-Tree index you'd code the `CREATE INDEX` Statement. When the column you're trying to index contains only a few distinct values as compared to the total number of rows in the table, a bitmap index may be your best bet! For example, let's create a bitmap index on the `GENDER` column on the `CANDYBAR_CONSUMPTION_DATA` table:

```
CREATE INDEX IX_CCD_GENDER
 ON TABLE CANDYBAR_CONSUMPTION_DATA(GENDER)
 AS 'BITMAP'
 WITH DEFERRED REBUILD;
```

Now, you can issue `SHOW INDEX ON table-name;` to see the indexes on the table (output modified to fit on the page):

```
beeline> show index on candybar_consumption_data;
+-----------------------+---------------------------+-----------------------+-----------------------+----------+
|       idx_name        |         tab_name          |       col_names       |       idx_type        | comment  |
+-----------------------+---------------------------+-----------------------+-----------------------+----------+
| ix_ccd_gender         | candybar_consumption_data | gender                | bitmap                |          |
+-----------------------+---------------------------+-----------------------+-----------------------+----------+
```

Now, since the `WITH DEFERRED BUILD` Clause was included, we must follow up with an `ALTER INDEX` Statement to rebuild the index. The general syntax looks like the following:

```
ALTER INDEX index_name ON table-name REBUILD;
```

For example, let's rebuild the index we created above:

```
ALTER INDEX IX_CCD_GENDER ON CANDYBAR_CONSUMPTION_DATA REBUILD;
```

Finally, you can drop an index simply by using the `DROP INDEX` Statement. For example,

```
DROP INDEX IX_CCD_GENDER ON CANDYBAR_CONSUMPTION_DATA;
```

Note that creating indexes is no longer available starting with Hive version 3 and above. This makes me sad! ☹ With this loss, though, the addition of materialized views and better storage formats more than makes up for it.

Partitioning Tables

HiveQL allows you to partition tables, but with fewer partitioning schemes as compared to ImpalaSQL. Please see *Chapter 16 – SQL Performance Improvements* for how to partition tables in ImpalaSQL. Also, see the HiveQL documentation for more on partitioning tables.

Computing Statistics

In *Chapter 6 – Introduction to SQL*, we briefly mentioned that statistics can be computed using the `COMPUTE STATS` Statement in ImpalaSQL. We describe this in more detail in *Chapter 16 – SQL Performance Improvements*. In HiveQL, the `ANALYZE TABLE` Statement plays a similar role. In general, the syntax looks like this:

```
ANALYZE TABLE table-name COMPUTE STATISTICS;
```

This flava-flav of the command computes statistics on the table as well as any associated partitions. If you'd like statistics to be computed on the columns as well, include the `FOR COLUMNS` Clause:

```
ANALYZE TABLE table-name COMPUTE STATISTICS FOR COLUMNS;
```

Normally, the `ANALYZE TABLE` Statement will compute statistics across all of the table's partitions. But, if you'd like to compute statistics on a specific partition, include the `PARTITION` Clause and specify the desired partition(s):

```
ANALYZE TABLE table-name PARTITION(partition-column-1=value-1,
                                   partition-column-2=value-2,
                                   ...
                                   partition-column-n=value-n)
 COMPUTE STATISTICS
 FOR COLUMNS;
```

For example, let's compute statistics on the table and columns of `CANDYBAR_CONSUMPTION_DATA`:

```
beeline> analyze table candybar_consumption_data
                                                compute statistics for columns;
INFO  : OK
No rows affected (39.507 seconds)
```

```
beeline>
```

If we created a table named `CANDYBAR_CONSUMPTION_DATA_PART` that was partitioned by the `GENDER` column, we could analyze a specific partition, say the males, like this:

```
analyze table candybar_consumption_data_part partition(gender='M')
 compute statistics
 for columns;
```

Chapter 15 – Complex Data Types in HiveQL and ImpalaSQL

HiveQL and ImpalaSQL allow for more than just the primitive data types (`TINYINT`, `STRING`, etc.) and include complex data types such as arrays, maps, structures, unions and combinations thereof. At this point, only `STORED AS PARQUET` tables containing complex data types are accessible from ImpalaSQL. Also, although you can access complex data types using ImpalaSQL, it's easier to work with them using HiveQL, in my opinion. (Note that you may want to skip this chapter upon first read of the book.)

All of the examples that follow were performed using the command line utility `beeline` to access the database and use HiveQL. With that said, occasionally you may receive a permissions error when using `beeline` with complex data types. Please work with your Hadoop Administrator to correct this issue. Although it's deprecated, you may still be able to use the `hive` command line utility in a pinch as a substitute for `beeline`. The permissions error doesn't seem to appear when using the `hive` command line utility.

Whether you can make use of complex data types in your SQL code remains to be seen, but they're nice to have in your back pocket just in case you need an alternative way of thinking beyond the standard SQL column/row drudgery.

The following are the **complex data types**, in brief:

- `ARRAY<`*`primitive-data-type_or_complex-data-type`*`>` – a zero-based array containing elements of either a single primitive data type or a complex data type
- `MAP<primitive-date-type-KEY,`*`primitive-data-type_or_complex-data-type_VALUE`*`>` – a map or dictionary containing key/value pairs where the key must be a primitive data type and the value can be either a primitive or complex data type
- `STRUCT<field-name:`*`primitive-data-type_or_complex-data-type`*`, … >` – a structure containing one or more field names and their associated primitive or complex data type values
- `UNIONTYPE<`*`primitive-data-type_or_complex-data-type`*`,`*`primitive-data-type_or_complex-data-type`*`, … >` – a union of different data types. This complex type is not fully supported yet, so we'll skip it for now.

Complex Data Type Constructors

HiveQL contains several functions which allow you to create complex data types by providing initializing values. Below is a list of the complex data type constructors:

- `ARRAY(`*`val1`*`,`*`val2`*`,…)` – this construtor returns an `ARRAY` complex data type based on the data type of *`val1`*, *`val2`*, etc. Note that *`val1`*, *`val2`*, etc. must be the same primitive or complex data type:

  ```
  SELECT ARRAY('A','B') AS COL1;

  +------------+--+
  |    col1    |
  +------------+--+
  | ["A","B"]  |
  +------------+--+
  ```

  The square brackets shown in the output above are an indicator of an array complex data type.

- `MAP(`*`key1`*`,`*`val1`*`,`*`key2`*`,`*`val2`*`,…)` – this constructor returns the `MAP` complex data type based on the data type of the keys and values. In the example below, `A`, `B` and `C` are the keys and `1`, `2` and `3` are the associated values:

  ```
  SELECT MAP('A',1,'B',2,'C',3) AS COL1;
  ```

```
+----------------------+--+
|         col1         |
+----------------------+--+
| {"A":1,"B":2,"C":3}  |
+----------------------+--+
```

The curly braces displayed in the output above are an indicator of a map complex data type.

- `STRUCT(`*`val1`*`,`*`val2`*`,…)` – this constructor returns a `STRUCT` complex data type based on the values *`val1`*, *`val2`*, etc. Each value will be associated with an automatically generated field name in the format `col#`:

  ```
  SELECT STRUCT('PA','PENNSYLVANIA') AS COL1;
  ```

  ```
  +-------------------------------------+--+
  |                col1                 |
  +-------------------------------------+--+
  | {"col1":"PA","col2":"PENNSYLVANIA"}  |
  +-------------------------------------+--
  ```

- `NAMED_STRUCT(`*`field-name1`*`,`*`val1`*`,`*`field-name2`*`,`*`val2`*`,…)` – this constructor is similar to the `STRUCT()` constructor above, but allows you to create your own field names rather than suffer with the automatically generated ghetto `col#` field names:

  ```
  SELECT NAMED_STRUCT('STATE_CODE','PA',
                      'STATE_NAME','PENNSYLVANIA') AS COL1;
  ```

  ```
  +-------------------------------------------------+--+
  |                       col1                      |
  +-------------------------------------------------+--+
  | {"state_code":"PA","state_name":"PENNSYLVANIA"}  |
  +-------------------------------------------------+--+
  ```

Complex Data Types from Primitive Data Types

There are several functions which return a complex data type using the primitive data types as input.

- `SPLIT(`*`string`*`,`*`delimiter`*`)` – this function splits a string based on the delimiter and places each piece into an `ARRAY` complex data type. For example,

  ```
  SELECT SPLIT('PA,NJ,DE,TX,ME',',') AS COL1;
  ```

  ```
  +----------------------------+--+
  |            col1            |
  +----------------------------+--+
  | ["PA","NJ","DE","TX","ME"]  |
  +----------------------------+--+
  ```

  The second argument to the `SPLIT()` function can be a regular expression as well.

- `STR_TO_MAP(`*`string`*`,`*`delimiter-1`*`,`*`delimiter-2`*`)` – this function creates a `MAP` complex data type. Note that *`delimiter-1`* is used to split the string into individual key/value pairs and *`delimiter-2`* then further splits each key/value pair into a key and a value. Below, the *`delimiter-1`* is a comma and *delimiter-2* is a colon. The function then separates the string into `A:1`, `B:2` and `C:3` and then further separates each into key and value: `A` with `1`, `B` with `2` and `C` with `3`:

```
SELECT STR_TO_MAP('A:1,B:2,C:3',',',':') AS COL1;

+---------------------------+--+
|           col1            |
+---------------------------+--+
| {"A":"1","B":"2","C":"3"}  |
+---------------------------+--+
```

Creating a Table with Complex Data Types

When creating a table using complex data types, you specify each column name along with its complex data type. This is probably not a surprise to you. For example,

```
CREATE TABLE COMPLEX_TYPES(aSTRINGTHINGS ARRAY<STRING>,
                           mDICTIONARY MAP<SMALLINT,STRING>,
                           aSTRUCT STRUCT<STATE_CODE:STRING,STATE_NAME:STRING>);
```

Naturally, a table can contain a mixture of primitive and complex data types…you're not stuck using just one or the other:

```
CREATE TABLE COMPLEX_TYPES(ROW_ID SMALLINT,
                           aSTRINGTHINGS ARRAY<STRING>,
                           mDICTIONARY MAP<SMALLINT,STRING>,
                           aSTRUCT STRUCT<STATE_CODE:STRING,STATE_NAME:STRING>);
```

Note that `aSTRINGTHINGS`, `mDICTIONARY` and `aSTRUCT` are the column names and the text appearing between the < and > symbols indicate the data type(s) used with the complex data types:

- `ARRAY<STRING>` indicates that the column `aSTRINGTHINGS` will contain an array of `STRING`s.
- `MAP<SMALLINT,STRING>` indicates that the column `mDICTIONARY` will contain a map with keys defined as `SMALLINT`s and values defined as `STRING`s.
- `STRUCT<STATE_CODE:STRING,STATE_NAME:STRING>` indicates that the column `aSTRUCT` will contain a structure with two fields `STATE_CODE` and `STATE_NAME` both of which are `STRING`s.

Note that `ARRAY`s and `MAP`s may contain more than just a single item whereas a `STRUCT` contains just a single item. That is, an `ARRAY` may contain all of the two-letter state codes, and a `MAP` may contain all of the two-letter state codes mapped to their associated state names. But, a `STRUCT`, as defined above, contains a single item: one two-letter state code and one state name. But, see the section *Combining Multiple Complex Types* below for more.

Inserting into a Table with Complex Data Types

You can insert data into a table containing complex data types in a variety of ways. In this section, we show you how to do this using a simplified `INSERT` Statement. In the section *Compressing Multiple Rows in a Table into a Complex Type*, we show you how to do this in a more automated fashion.

To insert data into an `ARRAY` complex data type, you can use the `ARRAY` constructor:

```
CREATE TABLE COMPLEX_TAB_1(ROW_ID SMALLINT,
                           aSTATECODES ARRAY<STRING>) STORED AS PARQUET;

INSERT INTO COMPLEX_TAB_1
 SELECT 1,ARRAY('AK','AL','PA','TX');

SELECT *
 FROM COMPLEX_TAB_1;
```

```
+-----------------------+----------------------------+--+
| complex_tab_1.row_id  | complex_tab_1.astatecodes  |
+-----------------------+----------------------------+--+
| 1                     | ["AK","AL","PA","TX"]      |
+-----------------------+----------------------------+--+
```

To insert data into a `MAP` complex data type, you can use the `MAP` constructor:

```
CREATE TABLE COMPLEX_TAB_2(ROW_ID SMALLINT,mDICTIONARY MAP<STRING,STRING>);

INSERT INTO COMPLEX_TAB_2
 SELECT 1,MAP('IA','MIDWEST',
              'IL','MIDWEST',
              'IN','MIDWEST',
              'KS','MIDWEST',
              'MI','MIDWEST',
              'MN','MIDWEST',
              'MO','MIDWEST',
              'ND','MIDWEST',
              'NE','MIDWEST');

SELECT *
 FROM COMPLEX_TAB_2;

+-----------------------+----------------------------------------+--+
| complex_tab_2.row_id  | complex_tab_2.mdictionary              |
+-----------------------+----------------------------------------+--+
| 1                     | {"IA":"MIDWEST","IL":"MIDWEST",...}    |
+-----------------------+----------------------------------------+--+
```

To insert data into a `STRUCT` complex data type, you can use either the `STRUCT` or `NAMED_STRUCT` construtors:

```
CREATE TABLE COMPLEX_TAB_3(ROW_ID INT,
                           aSTRUCT
                                  STRUCT<STATE_CODE:STRING,STATE_NAME:STRING>);

INSERT INTO COMPLEX_TAB_3
 SELECT 0,
        NAMED_STRUCT('state_code','ZZ','state_name','ZORBA');

SELECT *
 FROM COMPLEX_TAB_3;

+-----------------------+--------------------------------------------+--+
| complex_tab_3.row_id  | complex_tab_3.astruct                      |
+-----------------------+--------------------------------------------+--+
| 0                     | {"state_code":"ZZ","state_name":"ZORBA"}   |
+-----------------------+--------------------------------------------+--+
```

Since a `STRUCT` contains a single item, unlike `ARRAY`s and `MAP`s, you can use something like the following to insert data from a table into a table containing a `STRUCT` complex data type:

```
SELECT ROW_NUMBER() OVER (ORDER BY STRUCT_COL) AS ROW_ID,STRUCT_COL
  FROM (
        SELECT NAMED_STRUCT('STATE_CODE',STATE_CODE,
                            'STATE_NAME',STATE_NAME) AS STRUCT_COL
         FROM DIM_US_STATE_MAPPING
       ) A;
```

```
+---------+-----------------------------------------------------------+--+
| row_id  |                        struct_col                            |
+---------+-----------------------------------------------------------+--+
| 1       | {"state_code":"AA","state_name":"U.S. ARMED FORCES - AMERICAS"} |
| 2       | {"state_code":"AE","state_name":"U.S. ARMED FORCES - EUROPE"}   |
| 3       | {"state_code":"AK","state_name":"ALASKA"}                       |
| 4       | {"state_code":"AL","state_name":"ALABAMA"}                      |
| 5       | {"state_code":"AP","state_name":"U.S. ARMED FORCES - PACIFIC"}  |
| 6       | {"state_code":"AR","state_name":"ARKANSAS"}                     |
| 7       | {"state_code":"AS","state_name":"AMERICAN SAMOA"}               |
| 8       | {"state_code":"AZ","state_name":"ARIZONA"}                      |
| 9       | {"state_code":"CA","state_name":"CALIFORNIA"}                   |
| 10      | {"state_code":"CO","state_name":"COLORADO"}                     |
...snip...
```

Now, when using `NAMED_STRUCT`s, the field names must match the field names on the `CREATE TABLE` Statement. Also, the `INSERT INTO VALUES` Statement does not work with the complex data type constructors.

Compressing Multiple Rows in a Table into an ARRAY Complex Data Type

There may be times when you want to take the rows spanning an entire column and place them into a complex data type like an `ARRAY`. You can use the `COLLECT_SET` and `COLLECT_LIST` functions to create an `ARRAY` from the data from one column across multiple rows:

- `COLLECT_SET(`*`column-name`*`)` – This function takes the values appearing across *`column-name`*, **deduplicates** the values and returns an `ARRAY` complex data type.
- `COLLECT_LIST(`*`column-name`*`)` – This function takes the values appearing across *`column-name`*, does **not deduplicate** the values and returns an `ARRAY` complex data type.

For example, let's store the deduplicated two-letter state codes into an `ARRAY`:

```
SELECT COLLECT_SET(STATE_CODE)
 FROM DIM_US_STATE_MAPPING;

+----------------------------------------------------+--+
|                        _c0                         |
+----------------------------------------------------+--+
| ["SA","WA","NT","AB","BC","MB","NB",...snip...]    |
+----------------------------------------------------+--+
```

Accessing ARRAY, MAP and STRUCT Elements in a SQL Query

Once you have a complex data type defined in a table, you can access the elements contained within an `ARRAY`, `MAP` or `STRUCT` using plain ol' SQL code.

To access the elements of an `ARRAY`, you follow the `ARRAY` name with square brackets containing the index of the element you want returned. Take note that arrays are zero-based. For example,

```
SELECT ROW_ID,aSTATECODES[0] AS STATE_CODE
 FROM COMPLEX_TAB_1;

+---------+-------------+--+
| row_id  | state_code  |
+---------+-------------+--+
| 1       | AK          |
+---------+-------------+--+
```

To access the value for a particular key in a `MAP`, you follow the `MAP` column name with square brackets containing the desired key:

```
SELECT 'NE' AS STATE_CODE,mDICTIONARY['NE'] AS REGION
 FROM COMPLEX_TAB_2;

+-------------+---------+--+
| state_code  | region  |
+-------------+---------+--+
| NE          | MIDWEST |
+-------------+---------+--+
```

To access the fields in a `STRUCT`, you can use *ye olde dot-notation*: *`column-name.field-name`*:

```
SELECT ROW_ID,aSTRUCT.STATE_CODE,aSTRUCT.STATE_NAME
 FROM COMPLEX_TAB_3;

+---------+-------------+-----------------------------------------+--+
| row_id  | state_code  |               state_name                |
+---------+-------------+-----------------------------------------+--+
| 0       | ZZ          | ZORBA                                   |
+---------+-------------+-----------------------------------------+--+
```

You can also use the constucts above in a `WHERE` Clause:

```
SELECT ROW_ID,aSTRUCT.STATE_CODE,aSTRUCT.STATE_NAME
 FROM COMPLEX_TAB_3
 WHERE aSTRUCT.STATE_CODE IN ('CT','PA','ZZ');
```

Using the Collection Functions

HiveQL comes with several functions useful when working with complex data types.

Given an existing `MAP` column, you can retrieve the **keys** as an `ARRAY` by using the `MAP_KEYS(`*`map_column-name`*`)` function:

```
SELECT MAP_KEYS(mDICTIONARY) AS KEYS
 FROM COMPLEX_TAB_2;

+-------------------------------------------------------+--+
|                         keys                          |
+-------------------------------------------------------+--+
| ["IA","IL","IN","KS","MI","MN","MO","ND",...snip...]  |
+-------------------------------------------------------+--+
```

Given an existing `MAP` column, you can retrieve the **values** as an `ARRAY` by using the `MAP_VALUES(`*`map_column-name`*`)` function:

```
SELECT MAP_VALUES(mDICTIONARY) AS VALUES
 FROM COMPLEX_TAB_2;

+-------------------------------------------------------+--+
|                        values                         |
+-------------------------------------------------------+--+
| ["MIDWEST","MIDWEST","MIDWEST","MIDWEST",...snip...]  |
+-------------------------------------------------------+--+
```

The SIZE() function can be used with the MAP and ARRAY complex data types to retrieve the total number of elements:

```
SELECT SIZE(mDICTIONARY) AS NBR_OF_KEYS
 FROM COMPLEX_TAB_2;

+--------------+--+
| nbr_of_keys  |
+--------------+--+
| 51           |
+--------------+--+
```

You can sort an ARRAY using the SORT_ARRAY() function:

```
SELECT SORT_ARRAY(aSTATECODES) AS aSTATECODES_ASC
 FROM COMPLEX_TAB_1;

+-----------------------+--+
|    astatecodes_asc    |
+-----------------------+--+
| ["AK","AL","PA","TX"]  |
+-----------------------+--+
```

For an ARRAY, to determine if an element exists within it, use the ARRAY_CONTAINS() function. The return type is BOOLEAN (true or false):

```
SELECT ARRAY_CONTAINS(aSTATECODES,'ZZ') AS ZZ_IN,
       ARRAY_CONTAINS(aSTATECODES,'TX') AS TX_IN
 FROM COMPLEX_TAB_1;

+--------+--------+--+
| zz_in  | tx_in  |
+--------+--------+--+
| false  | true   |
+--------+--------+--+
```

Turning a Complex Data Type Back into Rows

Given an ARRAY or MAP column in a table, you can turn the complex data type back into columns/rows using the EXPLODE() function:

```
SELECT EXPLODE(mDICTIONARY)
 FROM COMPLEX_TAB_2;

+------+------------+--+
| key  |   value    |
+------+------------+--+
| IA   | MIDWEST    |
| IL   | MIDWEST    |
| IN   | MIDWEST    |
| KS   | MIDWEST    |
| MI   | MIDWEST    |
| MN   | MIDWEST    |
...snip...
```

For an ARRAY, if you'd like to display the array index numbers along with its data, you can use POSEXPLODE() function with a column alias containing a parenthesized list of column names:

```
SELECT POSEXPLODE(aSTATECODES) AS (INDX,STATE_CODE)
 FROM COMPLEX_TAB_1;

+--------+-------------+--+
| indx   | state_code  |
+--------+-------------+--+
| 0      | AK          |
| 1      | AL          |
| 2      | PA          |
| 3      | TX          |
+--------+-------------+--+
```

One *el cheap-o* use of the `POSEXPLODE()` function is to generate row numbers. In the code below, change the `20` – indicating the number of rows to be returned – to the number of rows you want generated:

```
SELECT A.RNBR
 FROM (
       SELECT POSEXPLODE(SPLIT(REPEAT("X`",20),"`")) AS (RNBR,COL)
      ) A
 WHERE A.RNBR>=1
 ORDER BY A.RNBR;

+---------+--+
| a.rnbr  |
+---------+--+
| 1       |
| 2       |
| 3       |
...snip...
| 18      |
| 19      |
| 20      |
+---------+--+
```

Naturally, you can use it with the `CREATE TABLE AS` Statement to create a table:

```
CREATE TABLE TEST_RNBR STORED AS PARQUET AS
 SELECT A.RNBR
  FROM (
        SELECT POSEXPLODE(SPLIT(REPEAT("X`",20),"`")) AS (RNBR,COL)
       ) A
  WHERE A.RNBR>=1;
```

Combining Multiple Complex Types

In the examples above, we only ever used a single complex data type at a time, but HiveQL allows you to intermingle multiple complex data types together. For example, let's create a table with a single column containing an `ARRAY` containing `STRUCT`s as its array elements:

```
CREATE TABLE COMPLEX_TAB_4(asSTUFF ARRAY<STRUCT<STATE_CODE:STRING,
                                                STATE_NAME:STRING>>);

+-----------+----------------------------------------------------+----------+--+
| col_name  |                     data_type                      | comment  |
+-----------+----------------------------------------------------+----------+--+
| asSTUFF   | array<struct<STATE_CODE:string,STATE_NAME:string>> |          |
+-----------+----------------------------------------------------+----------+--+
```

In order to populate this table, we need to create a `STRUCT` from the `STATE_CODE` and `STATE_NAME`, then use `COLLECT_LIST()` to force in all of the rows into the `ARRAY`:

```
SELECT COLLECT_LIST(mySTRUCT) AS myARRAY
 FROM (
       SELECT NAMED_STRUCT('STATE_CODE',A.STATE_CODE,
                           'STATE_NAME',A.STATE_NAME) AS mySTRUCT
        FROM (
              SELECT STATE_CODE,STATE_NAME
               FROM DIM_US_STATE_MAPPING
             ) A
      ) B;
```

Note that, in the output below, each `{}` indicates a named structure and the `[]` indicates the array:

```
+------------------------------------------------------------------------+--+
|                       myarray                                          |
+------------------------------------------------------------------------+--+
| [{"state_code":"AK","state_name":"ALASKA"},...snip...]                 |
+------------------------------------------------------------------------+--+
```

Finally, you can use the `INSERT` Statement with this code as well:

```
INSERT INTO COMPLEX_TAB_4
 SELECT COLLECT_LIST(mySTRUCT) AS myARRAY
  FROM (
        SELECT NAMED_STRUCT('STATE_CODE',A.STATE_CODE,
                            'STATE_NAME',A.STATE_NAME) AS mySTRUCT
         FROM (
               SELECT STATE_CODE,STATE_NAME
                FROM DIM_US_STATE_MAPPING
              ) A
       ) B;
```

Chapter 16 – SQL Performance Improvements

Although we've chatted about how to create and drop tables, insert data into tables, update tables and so on, we've avoided talking about improving the performance of your SQL queries. In this chapter, we discuss how to use partitions, improve `INSERT INTO` speeds, and much, much more.

Speeding Up Queries by Computing Statistics

Probably the best way to decrease query runtimes – other than partitioning the table appropriately, as we describe below – is to compute statistics on your tables. After you create a table, insert data into it, or any other change, you should issue the following command:

```
COMPUTE STATS prod_schema.table_name;
```

It shouldn't be a surprise that when you gather statistics on your tables, the query engine can determine the best possible plan of attack to run your query. Without up-to-date statistics, the query engine attempts to make a *best guess* plan of attack which may result in you staring at a hypnotic spinning icon in your SQL client's GUI interface for quite a wh…*zzzzzzzzzzzzzzzzzz*…

In addition to `COMPUTE STATS`, ImpalaSQL also has the `COMPUTE INCREMENTAL STATS` command, but that's mainly for partitioned tables. We discuss partitioning below. Note that if you accidentally use `COMPUTE INCREMENTAL STATS` on an unpartitioned table, it's equivalent to `COMPUTE STATS`. No wars will break out in Upper Volta.

Keeping Your Metadata Clean

Recall in *Chapter 4 – A Teensy-Weensy Chat about Hadoop*, we discussed the `INVALIDATE METADATA` Statement and indicated that when creating a table in Hive with HiveQL, Impala won't recognize its existence until you execute an `INVALIDATE METADATA` Statement on that table in ImpalaSQL. And this is all true, but there's more going on with metadata than that.

Now, suppose you're the type of person who buys boxes of donuts and disperses them about the house. (Nobody's judging you.) For example, a box of Krispy Kreme donuts is in the master bathroom. A Dunkin' Donuts box is placed in the living room. And a box of rat-assed store-bought donuts is placed in the basement. Now, your mind knows exactly where each box is located at any point in time and you can run, not walk, to the exact spot the donuts are located whenever your body's sugar levels ebb. But, suppose you run to the master bathroom and the box of Krispy Kreme donuts is missing! *Duhn-duhn!* What would you do? Well, any true-blooded donut afficionado would, clearly, run panic-stricken from room to room looking for the missing Krispy Kreme donuts until such a time as the missing box is found. This, of course, would cost calories! More importantly, it would cost valuable **time**. But, once you've found the box of Krispy Kreme donuts, your mind would make a note as to its new location, so next time you can head directly and speedily there. Well, this is eerily similar to how blocks of data are spread across HDFS. (I'm not sure the creators of Apache Hadoop ever envisioned this particular explanation for metadata as it pertains to blocks of data.)

When attempting to locate a block of data (i.e., box of donuts) and it's not where it was placed last time (e.g., master bathroom), you receive this very boring warning message:

```
WARNING: Read # MB of data across network that was expected to be
local. Block locality metadata for table 'schema.table' may be stale.
This only affects query performance and not result correctness. One of
the common causes for this warning is the HDFS rebalancer moving some
of the file's blocks. If this issue persists, consider running
"INVALIDATE METADATA `schema`.`table`".
```

As the message above indicates, the results will still be correct, but performance may suffer. And, if the table in question is your own table, then run `INVALIDATE METADATA` on it. If the table was created by someone else, I'd have a very stern conversation with them about donuts.

Now, suppose a neighbor concerned with your well-being purchases a fresh box of donuts and places it in one of the rooms in your house. (How the neighbor got into your house is a matter for the police.) Now, you'd never know that this fresh box of donuts even existed unless your neighbor kindly told you where it was located. You'd then make a mental note of its location. This is eerily similar to when a new data file is placed under an existing table's HDFS directory. You'd never know, and more importantly, Impala would never know unless you issue the `REFRESH` Statement on the associated table. At this point, Impala recognizes the new data file and updates the associated metadata on the table. Thus, when querying the table, the freshest donuts…uh, data…is picked up.

So, in brief, and I probably should have led with this, use:

- `INVALIDATE METADATA table-name;` when you've created the table, deleted one or more underlying files in HDFS associated with the table, inserted data into the table, deleted rows from the table, created the table in Hive and want to access it in Impala, etc.
- `REFRESH table-name;` when you've added one or more files into the HDFS directory associated with the table.

Are we all hankering for donuts now?

Speeding Up `INSERTs`

Based on the ImpalaSQL syntax for the `INSERT` Statement, you can provide multiple rows within a single `INSERT` Statement. Thus, you don't have to create a single `INSERT INTO VALUES` Statement line for each row you want to insert into a table. Rather, you create a single `INSERT INTO VALUES` Statement and provided a comma-delimited list of the rows.

For example, here's what everyone normally does:

```
INSERT INTO MYTABLE VALUES(1,2,3);
INSERT INTO MYTABLE VALUES(4,5,6);
INSERT INTO MYTABLE VALUES(7,8,9);
```

This works fine, but takes a long time to run if you have a significant number of inserts to perform. To speed up the inserts, you can use the following ImpalaSQL syntax instead:

```
INSERT INTO MYTABLE VALUES(1,2,3),
                          (4,5,6),
                          (7,8,9);
```

Just how fast we talkin' here? An insert with over `2000` individual `INSERT INTO VALUES` statements took about `6` minutes to complete…that's `6` minutes that could've been put to better use eating baked goods!! Now, using the syntax shown above, the `2000` inserts completed nearly **instantaneously**!! Schwing!!

Saving Space with `COMPRESSION_CODEC`

This is a more complicated topic than you might think at first blush. Recall that, in this book, we're concentrating on only three storage formats: `TEXTFILE`, `PARQUET` and `KUDU`. These three storage formats, as you can well imagine, store their underlying data in wildly different formats on disk. Despite that, ImpalaSQL allow you to compress the underlying data in HDFS to save space on disk at the cost of additional CPU processing time necessary to decompress the compressed data when needed to fulfill a query. You can't have one without it affecting the other. So, there's that problem, kids.

Now, recall that when you compress a file on your laptop using WinZip, you get back a single compressed file with a `.zip` extension. And, there's nothing wrong with that because you now have a single compressed file you can kick over to your backup drive or to cloud storage. Excellent! But, imagine if that same compressed file contains data needed to run a SQL query. How would a query engine handle processing that lone compressed file? Since it's a single file, only one CPU can handle working with it, despite having several CPUs sitting there staring at you blankly waiting for something useful to do. So, there's that problem, kids.

Okay, let's move away from your laptop/WinZip and head back to Hadoop terrain. Some compression formats allow a compressed `TEXTFILE`/`PARQUET`/`KUDU` file to be *splittable*; that is, despite being compressed, the file can be broken apart and handled by multiple CPUs (or processes, or threads…you get the point) allowing your query to complete faster. So, there's that problem, kids.

In order to tell ImpalaSQL which compression you fancy, you specify the `COMPRESSION_CODEC` option near the top of your SQL code:

```
SET COMPRESSION_CODEC=codec_option;
```

There are several available compression options you can choose from for *`codec_option`*, and a few of them are:

- ☐ `bread`
- ☐ `eggs`
- ☐ `cheese`

Oh, sorry, that's my shopping list. Here's the list of compression options:

- ☐ `snappy` – a general-purpose compression algorithm balancing compression size with the amount of processing time needed to decompress. Ahhh…like having your cake and eating it, too.
- ☐ `gzip` – much more compression than snappy, but with higher CPU decompression times.
- ☐ `none` – does not perform any compression and that makes me sad. ☹ Actually, if you're writing textual data to disk using `STORED AS TEXTFILE` specifically to give to someone, this is probably the way to go. See the example in *Chapter 1 – Quick Start Guide*.

For example, to turn on the `snappy` compression, execute the following code near the top of your SQL query:

```
SET COMPRESSION_CODEC=snappy;
```

Some general comments/recommendations:

1. By default, Impala compresses the `PARQUET` storage format with `snappy`. I would still provide the `SET COMPRESSION_CODEC=snappy;` line above in case the default format changes.
2. The `KUDU` storage format allows for column compression using `snappy`. I would still provide the `SET COMPRESSION_CODEC=snappy;` line above in case the default format changes.
3. Based on the internal organization of both the `PARQUET` and `KUDU` storage formats, the `snappy` compressed data may be splittable and can be processed concurrently.
4. When loading textual data using the `TEXTFILE` storage format, don't compress the file and decompress it if it's been delivered compressed. Load the data into the database (as described further below), promptly create a `snappy` compressed `PARQUET` table from it, and then drop the `TEXTFILE` table into oblivion.

Using the SHUFFLE Hint

As you may be aware, some databases provide several query or join *hints* which can be used with SQL queries in the hope of speeding up the queries. For example, in Oracle, to append data to a table faster, you can use the `APPEND` hint:

```
INSERT /*+ APPEND */ INTO ...
```

Now, I've found that by using the SHUFFLE hint in some ImpalaSQL queries, you can get huge performance gains!

But, before using this hint, ensure that you've computed the appropriate stats on the tables involved in the query. Check the query's runtime again to see if your query is faster. If not, try the SHUFFLE hint. Fingers crossed!!

In order to use SHUFFLE, you should also use the keyword STRAIGHT_JOIN in your query. For example,

```
SELECT STRAIGHT_JOIN
       A.COLUMN_1,
       B.COLUMN_2,
       C.COLUMN_3
  FROM TABLE_A A LEFT JOIN TABLE_B B
  ON A.COLUMN_1 = B.COLUMN_2
   LEFT JOIN TABLE_C C
   ON A.COLUMN_1 = C.COLUMN_1
    LEFT JOIN TABLE_D D
    ON C.COLUMN_2 = D.COLUMN_2
     LEFT JOIN /* +SHUFFLE */ TABLE_E E
     ON A.COLUMN_1 = E.COLUMN_1;
```

Here's some 'splainin':

- STRAIGHT_JOIN – this prevents the SQL optimizer from switching the order of the table names as they appear in the SQL code; that is, the order of the tables shown in the SQL code above (TABLE_A, TABLE_B, TABLE_C, TABLE_D, TABLE_E) is the order in which the code is performed.
- /* +SHUFFLE */ *table_name* – this prevents Hadoop from copying all of the data across the network to all of the nodes involved in the query. This option will perform a *hashing function* on the join columns and only the corresponding data is sent to the appropriate nodes, not all of it. The keyword /* +SHUFFFLE */ must **precede the table** you want to shuffle across to the nodes. The reason STRAIGHT_JOIN is used is because we want the shuffle to be associated with the table we want, and not get lost if Impala optimizes our SQL by changing the order of the tables. In the example above, TABLE_E is shuffled across the network.
- /* +BROADCAST */ *table_name* – By default, Hadoop sends all of the data from each table across the network, which is probably something you want to avoid unless your tables are very small.

Now, a hashing function is similar to how, say, you look up a surname in a telephone book. For example, a surname beginning with the letter 'A' is mapped to the hash 'A'; the letter 'B', to the hash 'B', etc. for 26 hashes. Thus, if you know the first letter of the surname, you can quickly go to that part of the telephone book and ignore the rest. And, less data being sent across the network is a good thing!

Using the SORT BY Statement

Most useful to tables stored using the Parquet format, the SORT BY Statement indicates how the data in the table should be sorted. The SORT BY Statement is best with the CTAS syntax or with a CREATE TABLE followed directly by an INSERT INTO. The target table will be sorted based on the columns specified in the SORT BY Statement. For example, let's re-create the DIM_POSTAL_CODE table such that it's sorted first by STATE_CODE and then by POSTAL_CODE:

```
CREATE TABLE DIM_POSTAL_CODE_SORTED(POSTAL_CODE STRING,
                                    CITY STRING,
                                    STATE_CODE STRING,
                                    LATITUDE DOUBLE,
                                    LONGITUDE DOUBLE)
 SORT BY (STATE_CODE,POSTAL_CODE)
 STORED AS PARQUET;
```

```
INSERT INTO DIM_POSTAL_CODE_SORTED
 SELECT *
  FROM DIM_POSTAL_CODE;

COMPUTE STATS DIM_POSTAL_CODE_SORTED;
```

Let's display the first few rows of `DIM_POSTAL_CODE_SORTED` as well as `DIM_POSTAL_CODE`:

```
[hdpserver.com:21000] prod_schema> SELECT *
                                    FROM DIM_POSTAL_CODE_SORTED
                                    LIMIT 20;
+-------------+-----------+------------+-----------+-------------+
| postal_code | city      | state_code | latitude  | longitude   |
+-------------+-----------+------------+-----------+-------------+
| 09323       | APO       | AE         | -44.25    | 33.53       |
| 99501       | ANCHORAGE | AK         | 61.216799 | -149.87828  |
| 99502       | ANCHORAGE | AK         | 61.153693 | -149.95932  |
| 99503       | ANCHORAGE | AK         | 61.19026  | -149.89341  |
| 99504       | ANCHORAGE | AK         | 61.204466 | -149.74633  |
| 99505       | JBER      | AK         | 61.261518 | -149.66336  |
| 99506       | JBER      | AK         | 61.224384 | -149.77461  |
| 99507       | ANCHORAGE | AK         | 61.154834 | -149.82865  |
| 99508       | ANCHORAGE | AK         | 61.203953 | -149.8144   |
| 99509       | ANCHORAGE | AK         | 61.108864 | -149.440311 |
| 99510       | ANCHORAGE | AK         | 61.144568 | -149.878418 |
| 99511       | ANCHORAGE | AK         | 61.068324 | -149.800476 |
| 99512       | ANCHORAGE | AK         | 61.203954 | -149.808426 |
| 99513       | ANCHORAGE | AK         | 61.214877 | -149.88617  |
| 99514       | ANCHORAGE | AK         | 61.108864 | -149.440311 |
| 99515       | ANCHORAGE | AK         | 61.122943 | -149.88852  |
| 99516       | ANCHORAGE | AK         | 61.101142 | -149.77311  |
| 99517       | ANCHORAGE | AK         | 61.188276 | -149.93438  |
| 99518       | ANCHORAGE | AK         | 61.156565 | -149.88335  |
| 99519       | ANCHORAGE | AK         | 61.108864 | -149.440311 |
+-------------+-----------+------------+-----------+-------------+

[hdpserver.com:21000] prod_schema> SELECT *
                                    FROM DIM_POSTAL_CODE
                                    LIMIT 20;
+-------------+-------------+------------+-----------+--------------------+
| postal_code | city        | state_code | latitude  | longitude          |
+-------------+-------------+------------+-----------+--------------------+
| 00623       | CABO ROJO   | PR         | 18.08643  | -67.15222          |
| 00633       | CAYEY       | PR         | 18.194527 | -66.18346699999999 |
| 00640       | COAMO       | PR         | 18.077197 | -66.359104         |
| 00676       | MOCA        | PR         | 18.37956  | -67.08423999999999 |
| 00728       | PONCE       | PR         | 18.013353 | -66.65218          |
| 00734       | PONCE       | PR         | 17.999499 | -66.643934         |
| 00735       | CEIBA       | PR         | 18.258444 | -65.65987          |
| 00748       | FAJARDO     | PR         | 18.326732 | -65.652484         |
| 00766       | VILLALBA    | PR         | 18.126023 | -66.48208          |
| 00771       | LAS PIEDRAS | PR         | 18.18744  | -65.87088          |
| 00791       | HUMACAO     | PR         | 18.147257 | -65.82268999999999 |
| 00901       | SAN JUAN    | PR         | 18.465426 | -66.10786          |
| 00906       | SAN JUAN    | PR         | 18.46454  | -66.10079          |
| 00909       | SAN JUAN    | PR         | 18.442282 | -66.06764          |
| 00922       | SAN JUAN    | PR         | 18.410462 | -66.06053300000001 |
| 00924       | SAN JUAN    | PR         | 18.401917 | -66.01194          |
| 00961       | BAYAMON     | PR         | 18.412462 | -66.16033          |
| 01704       | FRAMINGHAM  | MA         | 42.446396 | -71.459405         |
```

```
| 01731       | HANSCOM AFB | MA         | 42.459085 | -71.27556          |
| 01746       | HOLLISTON   | MA         | 42.196065 | -71.43797000000001 |
+-------------+-------------+------------+-----------+--------------------+
```

As you see, the sorted table is sorted by `STATE_CODE` first and `POSTAL_CODE` within `STATE_CODE`.

Note that, if you're creating a partitioned table, don't specify the partitioning columns on the `SORT BY` Statement.

Caching Tables

Although only available with the latest versions of Impala, you can pin small dimension tables directly in memory across the cluster. This prevents your dimension tables from being read from disk into memory since they're already there.

Note that your mage-like Hadoop Administrator will have to create a *cache pool name* for you to use with your dimension tables. Now, there are several ways to place a dimension table in memory:

- ☐ `CREATE TABLE...CACHED IN 'pool-name'` – When you create a dimension table anew, you can place it into memory by providing the `CACHED IN` Clause after the `STORED AS` Clause.
- ☐ `ALTER TABLE table-name SET CACHED IN 'pool-name'` – If the table already exists, it can be pinned to memory by using the `CACHED IN` Clause of the `ALTER TABLE` Statement.
- ☐ `CREATE TABLE table-name CACHED IN 'pool-name' AS select-statement` – You can use the CTAS syntax along with the `CACHED IN` Clause to pin a dimension table to memory.

If you think this is something you'd like to pursue, please have a conversation with your high-IQ Hadoop Administrator.

Working with Partitions – General Comments

Before the partition-specific syntax hullabaloo is shown, let's have a brief discussion about partitions:

- ☐ The act of partitioning a table is the physical separation of a table's data into smaller slices – or *partitions* – of a table's data.
- ☐ A table can be partitioned using one or more of the columns appearing in the table. For example, the table `DIM_POSTAL_CODE` contains the two-letter `STATE_CODE` column which could be used to partition the table into `61` individual partitions, one for each two-letter state code appearing under the `STATE_CODE` column.
- ☐ Each one of the partitions is stored as a subdirectory under the table's own HDFS **directory**. For example, as we've seen, the table `DIM_POSTAL_CODE` is located in the HDFS directory `hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_postal_code`. If partitioned by the column `STATE_CODE`, `61` **subdirectories** would be created. For example, the full directory name for the **North Carolina** partition would appear as follows:

  `hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_postal_code/`**`state_code=NC`**

- ☐ When working with a table that's partitioned, you never have to worry about the named subdirectories in HDFS, just like working with a table that's not partitioned. In other words, just use the table name in your SQL code and let Hadoop/Impala do the dirty work.
- ☐ The comment above is a teensy-weensy bald-faced lie. When inserting data into a partitioned table, you must ensure the partitioning column(s) are indicated, either hard-coded or not. We talk more about this below.
- ☐ The total number of rows across all of the partitions of a table *usually* equals the total number of rows in the unpartitioned table. *Usually* because you don't have to load all of the data into partitions and some partitions can be ignored. For example, real estate not part of the 48 contiguous states can be ignored, such as `AK`, `HI`, `AE`, `FM`, `PW`, etc.

- ☐ Once a table is partitioned, Impala can skip over the partitions not being visited by your SQL query. This is known as *partition pruning* and occurs no matter whether you've hard-coded values in your `WHERE` Clause (*static partition pruning*) or partitions are picked up from your behemoth SQL query at runtime (*dynamic partition pruning*). For example, if your SQL query is concerned with pulling data from North Carolina and South Carolina (`WHERE STATE_CODE  IN ('NC','SC')`), the remaining partitions will be ignored.
- ☐ Once a table is partitioned, Impala *may* process your query in parallel. Similar to other database engines, Impala uses computed statistics to determine if running a query in parallel would be beneficial. If so, the query's then broken up and processed concurrently and the final individual pieces will be smooshed together to produce one big smooshy result.
- ☐ Not every table needs to be partitioned. For example, the table `DIM_US_STATE_MAPPING` contains the two-letter US `STATE_CODE` along with corresponding name in `STATE_NAME`. This table contains only `65` rows, one row per two-letter state code, and wouldn't necessarily benefit from partitioning by `STATE_CODE`. A table this size can be broadcast easily enough across the network without the network cables and motherboard traces igniting.
- ☐ Every table should have fresh statistics computed on it regardless whether it's partitioned or not. As we've seen above, the `COMPUTE STATISTICS` command is used to compute the statistics for the table. An alternative to `COMPUTE STATISTICS` is `COMPUTE INCREMENTAL STATISTICS` which is used on partitioned tables. When using `COMPUTE STATISTICS`, any prior computed statistics are flushed and replaced wholesale regardless if the table is partitioned or not. When adding a new partition to a table (the `51`st state of `CL=CANDYLAND` in our example?), and you've already computed statistics on the previous `50` partitions, use `COMPUTE INCREMENTAL STATISTICS` to compute stats only on the new partition. We describe this in more detail below.

With that said, the `$1,000,000` question is: *How can a regular person like me determine which columns should be used to partition a table?* The easiest way to determine this is by looking at the `WHERE` Clauses in your SQL code. If you frequently use, say, `WHERE STATE_CODE='NC'` or `WHERE STATE_CODE IN (...)`, then `STATE_CODE` may be a good candidate for a partition column. If you frequently subset your table by, say, the year/month column `YYYYMM` (`WHERE YYYYMM BETWEEN 202001 AND 202012`), maybe partitioning by `YYYYMM` may be appropriate.

With that said, partitioning a table is a delicate balance between **too many partitions with way too little data** versus **too few partitions and way too much data**. Too many partitions and concurrency becomes the bottleneck. Too much data and processing becomes the bottleneck. You may want to try different partitioning schemes until you've found one that's appropriate.

Note that you can use your legacy database, if appropriate, as a guide here. If the table in question is partitioned a certain way in the legacy database, you may want to use that partitioning scheme in Hadoop as well. Be aware that your legacy database may have more methods to partition a table than are available in Hadoop.

Finally, if you're still having difficulty determining how to partition a table, please have a conversation with your handy-dandy Hadoop Administrator.

Working with Partitions (TEXTFILE/PARQUET)

When creating a partitioned table with the `CREATE TABLE` Statement, you include the `PARTITIONED BY` Clause to indicate your desired partitioning scheme. When altering a `TEXTFILE` or `PARQUET` partitioned table with the `ALTER TABLE` Statement, you use `ALTER TABLE`'s `PARTITION` Clause to modify the partitioning scheme.

As a quick example, let's create the partitioned table `DIM_POSTAL_CODE_PART` using data from the non-partitioned table `DIM_POSTAL_CODE`. Below is the `CREATE TABLE` syntax for the partitioned table:

```
CREATE TABLE DIM_POSTAL_CODE_PART(
 POSTAL_CODE  STRING,
 CITY         STRING,
 LATITUDE     DOUBLE,
 LONGITUDE    DOUBLE
)
```

```
PARTITIONED BY (
 STATE_CODE STRING
)
COMMENT 'DIM_POSTAL_CODE PARTITIONED BY STATE_CODE'
STORED AS PARQUET
TBLPROPERTIES('transactional'='false');
```

Note that the column `STATE_CODE` has been moved from its spot in the column definitions into the `PARTITIONED BY` Clause along with its data type.

In the `CREATE TABLE` Statement above, the table property `transactional=false` is included to allow the `ALTER TABLE` Statement, which we use below, to function properly. Without setting `transactional=false`, the `ALTER TABLE` Statement fails issuing the following adorable message:

```
ERROR: AnalysisException: ALTER TABLE not supported on transactional
(ACID) table: prod_schema.dim_postal_code_part.
```

Now, just like its non-partitioned brother, the table `DIM_POSTAL_CODE_PART` has a corresponding directory in HDFS:

```
hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_postal_code_part
```

At this point, no subdirectories appear under this directory because the table is as empty as yesterday's box of donuts. Let's fix that now by using the `INSERT` Statement:

```
INSERT INTO DIM_POSTAL_CODE_PART(POSTAL_CODE,
                                 CITY,
                                 LATITUDE,
                                 LONGITUDE,
                                 STATE_CODE)
 SELECT POSTAL_CODE,CITY,LATITUDE,LONGITUDE,STATE_CODE
  FROM DIM_POSTAL_CODE;
```

Note that I'm specifying the list of column names in parentheses after the table name. You must ensure that the partitioning columns appear last. For example, since `STATE_CODE` is our partitioning column, it must appear last in the column list. This ensures that Impala won't yell at you because it thinks you're missing the partitioning column.

After inserting data into the table, we need to compute incremental statistics for all of the partitions. Before doing that, let's display useful information about the partitions using the `SHOW PARTITIONS` Statement (output edited to fit on the page):

```
[hdpserver.com:21000] prod_schema> SHOW PARTITIONS DIM_POSTAL_CODE_PART;
+------------+-------+--------+---------+--------------------------------------+
| state_code | #Rows | #Files | Size    | Location                             |
+------------+-------+--------+---------+--------------------------------------+
| AE         | -1    | 1      | 1.14KB  | .../dim_postal_code_part/state_code=AE |
| AK         | -1    | 1      | 9.96KB  | .../dim_postal_code_part/state_code=AK |
...snip...
| WV         | -1    | 1      | 29.84KB | .../dim_postal_code_part/state_code=WV |
| WY         | -1    | 1      | 7.61KB  | .../dim_postal_code_part/state_code=WY |
| Total      | -1    | 61     | 1.31MB  |                                      |
+------------+-------+--------+---------+--------------------------------------+
```

Note that the column `#Rows` displays a `-1` indicating that stats have not been calculated on this table yet. Let's compute stats now:

```
COMPUTE INCREMENTAL STATS DIM_POSTAL_CODE_PART;
```

And, let's see the partition information again:

```
[hdpserver.com:21000] prod_schema> SHOW PARTITIONS DIM_POSTAL_CODE_PART;
+------------+-------+--------+---------+--------------------------------------+
| state_code | #Rows | #Files | Size    | Location                             |
+------------+-------+--------+---------+--------------------------------------+
| AE         | 1     | 1      | 1.14KB  | .../dim_postal_code_part/state_code=AE |
| AK         | 278   | 1      | 9.96KB  | .../dim_postal_code_part/state_code=AK |
...snip...
| WV         | 943   | 1      | 29.84KB | .../dim_postal_code_part/state_code=WV |
| WY         | 203   | 1      | 7.61KB  | .../dim_postal_code_part/state_code=WY |
| Total      | 43689 | 61     | 1.31MB  |                                      |
+------------+-------+--------+---------+--------------------------------------+
```

As you see, the column `#Rows` is filled in.

Note that, regardless if your table is partitioned or not, you can alternatively use `SHOW TABLE STATS` to display similar statistics to those shown above.

```
[hdpserver.com:21000] prod_schema> SHOW TABLE STATS DIM_POSTAL_CODE;
+-------+--------+--------+--------------+------------+---------+------------+---------------------+
| #Rows | #Files | Size   | Bytes Cached | Cache Repl | Format  | Incr stats | Location            |
+-------+--------+--------+--------------+------------+---------+------------+---------------------+
| 43689 | 1      | 1.31MB | NOT CACHED   | NOT CACHED | PARQUET | false      | .../dim_postal_code |
+-------+--------+--------+--------------+------------+---------+------------+---------------------+
```

But, running `SHOW PARTITIONS` on a table that is **not** partitioned will yield the following sadness:

```
[hdpserver.com:21000] prod_schema> SHOW PARTITIONS DIM_POSTAL_CODE;
ERROR: AnalysisException: Table is not partitioned: prod_schema.dim_postal_code
```

Now, in Hadoop, let's see what's under the `dim_postal_code_part` directory in HDFS:

```
[smithbob@lnxserver ~]$ hadoop fs -ls
                            hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_postal_code_part
Found 61 items
drwxrwx---+  - impala hive          0 2022-04-04 13:42
            hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_postal_code_part/state_code=AE
drwxrwx---+  - impala hive          0 2022-04-04 13:42
            hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_postal_code_part/state_code=AK
...snip...
drwxrwx---+  - impala hive          0 2022-04-04 13:42
            hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_postal_code_part/state_code=WV
drwxrwx---+  - impala hive          0 2022-04-04 13:42
            hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_postal_code_part/state_code=WY
```

Next, let's use the `EXPLAIN` Statement to see if partition pruning is having an effect on our query. The syntax for the `EXPLAIN` Statement is very straightforward:

```
EXPLAIN select-statement;
```

First, here's a SQL query that goes up against the non-partitioned table:

```
[hdpserver.com:21000] prod_schema> explain select * from dim_postal_code where
state_code in ('NJ','PA');
+-----------------------------------------------------------+
| Explain String                                            |
+-----------------------------------------------------------+
| Max Per-Host Resource Reservation: Memory=2.00MB Threads=3 |
| Per-Host Resource Estimates: Memory=80MB                  |
| Codegen disabled by planner                               |
|                                                           |
| PLAN-ROOT SINK                                            |
| |                                                         |
| 01:EXCHANGE [UNPARTITIONED]                               |
| |                                                         |
| 00:SCAN HDFS [prod_schema.dim_postal_code]                |
```

```
|    HDFS partitions=1/1 files=1 size=1.28MB                    |
|    predicates: state_code IN ('NJ', 'PA')                     |
|    row-size=68B cardinality=1.43K                             |
+---------------------------------------------------------------+
```

And, here's a SQL query that goes up against the partitioned table now:

```
[hdpserver.com:21000] prod_schema> explain select * from dim_postal_code_part
where state_code in ('NJ','PA');
+-----------------------------------------------------------------+
| Explain String                                                  |
+-----------------------------------------------------------------+
| Max Per-Host Resource Reservation: Memory=128.00KB Threads=3    |
| Per-Host Resource Estimates: Memory=64MB                        |
| Codegen disabled by planner                                     |
|                                                                 |
| PLAN-ROOT SINK                                                  |
| |                                                               |
| 01:EXCHANGE [UNPARTITIONED]                                     |
| |                                                               |
| 00:SCAN HDFS [prod_schema.dim_postal_code_part]                 |
|    partition predicates: state_code IN ('NJ', 'PA')             |
|    HDFS partitions=2/61 files=2 size=96.41KB                    |
|    row-size=66B cardinality=3.05K                               |
+-----------------------------------------------------------------+
```

In the output directly above, only 2 partitions of the 61 are being accessed! Nice!! In the first output, since there's effectively only one partition (the entire bloody table), the query trundles through all of the data like a lugubrious trundling thing.

Now, let's use the ALTER TABLE Statement to add a new partition for the great state of CL (CANDYLAND):

```
ALTER TABLE DIM_POSTAL_CODE_PART ADD IF NOT EXISTS PARTITION (STATE_CODE='CL');
```

In this case, we're hardcoding the name of the partition, CL. As you would expect, the column #Rows in the SHOW PARTITIONS output is set to -1 for the CL partition. Although the previous output from SHOW PARTITIONS was redacted, the useful column Incremental Stats is included this time:

```
[hdpserver.com:21000] prod_schema> SHOW PARTITIONS DIM_POSTAL_CODE_PART;
+------------+-------+--------+---------+-------------------+--------------------------------------+
| state_code | #Rows | #Files | Size    | Incremental Stats | Location                             |
+------------+-------+--------+---------+-------------------+--------------------------------------+
| AE         | 1     | 1      | 1.14KB  | true              | .../dim_postal_code_part/state_code=AE |
| AK         | 278   | 1      | 9.96KB  | true              | .../dim_postal_code_part/state_code=AK |
...snip...
| CL         | -1    | 0      | 0B      | false             | .../dim_postal_code_part/state_code=CL |
...snip...
| WV         | 943   | 1      | 29.84KB | true              | .../dim_postal_code_part/state_code=WV |
| WY         | 203   | 1      | 7.61KB  | true              | .../dim_postal_code_part/state_code=WY |
| Total      | 43689 | 61     | 1.31MB  | true              |                                      |
+------------+-------+--------+---------+-------------------+--------------------------------------+
```

Next, let's insert some fake data into that one partition using the PARTITION Clause of the INSERT Statement:

```
INSERT INTO DIM_POSTAL_CODE_PART PARTITION(STATE_CODE='CL')
 SELECT POSTAL_CODE,CITY,LATITUDE,LONGITUDE
  FROM DIM_POSTAL_CODE
  WHERE STATE_CODE='PA';
```

After this code completes, the following message is displayed:

```
Modified 2287 row(s) in 0.21s
```

Note that 2287 is the same number of rows for Pennsylvania, of course. Showing the partition information again yields the following:

```
[hdpserver.com:21000] prod_schema> SHOW PARTITIONS DIM_POSTAL_CODE_PART;
+------------+-------+--------+---------+-------------------+--------------------------------------+
| state_code | #Rows | #Files | Size    | Incremental Stats | Location                             |
+------------+-------+--------+---------+-------------------+--------------------------------------+
| AE         | 1     | 1      | 1.14KB  | true              | .../dim_postal_code_part/state_code=AE |
| AK         | 278   | 1      | 9.96KB  | true              | .../dim_postal_code_part/state_code=AK |
...snip...
| CL         | -1    | 1      | 72.20KB | false             | .../dim_postal_code_part/state_code=CL |
...snip...
| WY         | 203   | 1      | 7.61KB  | true              | .../dim_postal_code_part/state_code=WY |
| Total      | 43689 | 62     | 1.38MB  | true              |                                      |
+------------+-------+--------+---------+-------------------+--------------------------------------+
```

Again, #Rows is set to -1 for CL. Now, let's compute incremental stats on this table so that the CL partition is all primped and ready for luv:

```
[hdpserver.com:21000] prod_schema> COMPUTE INCREMENTAL STATS
                                                    DIM_POSTAL_CODE_PART;
+-----------------------------------------+
| summary                                 |
+-----------------------------------------+
| Updated 1 partition(s) and 4 column(s). |
+-----------------------------------------+
```

And, the informational message displayed above indicates that only one partition was updated. Let's see the partitions now:

```
[hdpserver.com:21000] prod_schema> SHOW PARTITIONS DIM_POSTAL_CODE_PART;
+------------+-------+--------+---------+-------------------+--------------------------------------+
| state_code | #Rows | #Files | Size    | Incremental Stats | Location                             |
+------------+-------+--------+---------+-------------------+--------------------------------------+
| AE         | 1     | 1      | 1.14KB  | true              | .../dim_postal_code_part/state_code=AE |
| AK         | 278   | 1      | 9.96KB  | true              | .../dim_postal_code_part/state_code=AK |
...snip...
| CL         | 2287  | 1      | 72.20KB | true              | .../dim_postal_code_part/state_code=CL |
...snip...
| WV         | 943   | 1      | 29.84KB | true              | .../dim_postal_code_part/state_code=WV |
| WY         | 203   | 1      | 7.61KB  | true              | .../dim_postal_code_part/state_code=WY |
| Total      | 43689 | 62     | 1.38MB  | true              |                                      |
+------------+-------+--------+---------+-------------------+--------------------------------------+
```

Now, the #Rows column is filled in and incremental stats is set to true. Huzzah!!

Hash Partitioning (TEXTFILE/PARQUET)

Based on the PARTITIONED BY syntax shown above for both the TEXTFILE and PARQUET storage formats, there's no pre-built method to create hash partitions, unlike for the KUDU storage format (described in more detail below). But, we're programmers, so let's *fake this tatertot out* (as the young kids say).

Recall that hash is a breakfast food generally consisting of small, yet manageable, bits of chopped up meats, cut up vegetables, diced potatoes and, as the Internet suggests: *anything else you have laying around that's still edible*, then cooked until ignited and lovingly finished with a dousing of Pepto Bismol.

The key here is the *small, yet manageable, bits*. This is, effectively, what hash partitioning is: take a large table that doesn't really have a useful partitioning column (e.g., year, year/month, country code, state code, etc.) and cut it up into small, yet manageable, bits.

Now, in order to do this, we make use of the ImpalaSQL function `fnv_hash()` which takes any argument and returns a `BIGINT` value based on the argument. [What follows makes use of the discussion of `fnv_hash()` in the ImpalaSQL manual, and isn't my own creation…credit where credit's due, pal!]
Since `fnv_hash()` returns both positive and negative values, we can use the `abs()` function to convert the returned values into positive values:

```
abs(fnv_hash(column-name))
```

We then follow up with the modulo operator (%) followed by the number of hash partitions we want:

```
abs(fnv_hash(column-name)) % nbr-partitions
```

Finally, depending on the range of values produced from the code fragment above, we can cast the value to the appropriate data type:

```
cast(abs(fnv_hash(column-name)) % nbr-partitions) as data-type)
```

For example, let's make use of the table `DIM_POSTAL_CODE` and create a hash partitioned version of it using the `LATITUDE` column.

```
USE PROD_SCHEMA;
CREATE TABLE DIM_POSTAL_CODE_HASH(POSTAL_CODE STRING,
                                  CITY STRING,
                                  STATE_CODE STRING,
                                  LATITUDE DOUBLE,
                                  LONGITUDE DOUBLE)
 PARTITIONED BY (PARTKEY TINYINT)
 STORED AS PARQUET;
```

Note that we're assigning the column `PARTKEY` as the partitioning column here which will be populated, along with the rest of the table's data, by the `INSERT` Statement below:

```
INSERT INTO DIM_POSTAL_CODE_HASH(POSTAL_CODE,
                                 CITY,
                                 STATE_CODE,
                                 LATITUDE,
                                 LONGITUDE,
                                 PARTKEY)
 SELECT A.POSTAL_CODE,A.CITY,A.STATE_CODE,A.LATITUDE,A.LONGITUDE,A.PARTKEY
  FROM (
        SELECT POSTAL_CODE,CITY,STATE_CODE,LATITUDE,LONGITUDE,
               CAST(ABS(FNV_HASH(LATITUDE)) % 10 AS TINYINT) AS PARTKEY
         FROM DIM_POSTAL_CODE
       ) A;
```

In the code above, we're creating `10` hash partitions ranging from `0` to `9`. Next, let's compute stats on the entire table:

```
COMPUTE STATS DIM_POSTAL_CODE_HASH;
```

And, let's see the partitions for the table:

```
[hdpserver.com:21000] default> SHOW PARTITIONS DIM_POSTAL_CODE_HASH;
+---------+-------+--------+----------+--------------------------------------------------------------+
| partkey | #Rows | #Files | Size     | Location                                                     |
+---------+-------+--------+----------+--------------------------------------------------------------+
| 0       | 4463  | 1      | 141.15KB | hdfs://lnxserver.com:8020/.../dim_postal_code_hash/partkey=0 |
| 1       | 4369  | 1      | 139.32KB | hdfs://lnxserver.com:8020/.../dim_postal_code_hash/partkey=1 |
...snip...
| 8       | 4434  | 1      | 142.00KB | hdfs://lnxserver.com:8020/.../dim_postal_code_hash/partkey=8 |
| 9       | 4255  | 1      | 137.69KB | hdfs://lnxserver.com:8020/.../dim_postal_code_hash/partkey=9 |
| Total   | 43689 | 10     | 1.37MB   |                                                              |
```

```
+---------+-------+--------+----------+----------------------------------------------------------+
```

Finally, we can query the table DIM_POSTAL_CODE_HASH, but since the partitioning column isn't a column we would normally use in a query, such as year, we have to include code to limit to the appropriate partitions as well, like this:

```
SELECT COUNT(*)
 FROM DIM_POSTAL_CODE_HASH
 WHERE LATITUDE=47.376884
       AND PARTKEY=CAST(ABS(FNV_HASH(LATITUDE)) % 10 AS TINYINT);
```

Working with Partitions (KUDU)

Although we indicated earlier that the whole primary key hullabaloo is unnecessary, that comment was specifically for the TEXTFILE and PARQUET storage formats. Since the KUDU storage format is used for UPDATEs and DELETEs, a primary key is essential here. Recall that a primary key can be made up of one or more columns.

Now, a table stored using the KUDU storage format can also be partitioned and, in order to make use of parallel processing, it probably should be partitioned. When creating a partitioned table specifying STORED AS KUDU with the CREATE TABLE Statement, you use the PARTITION BY Clause. When altering a KUDU partitioned table with the ALTER TABLE Statement, you use ALTER TABLE's PARTITION Clause.

The KUDU storage format allows for both *hash* and *range* partitioning as well as a combination thereof. Contrasting this with tables using the PARQUET storage format and its equality-style partitioning, the KUDU storage format allows for more unique partitioning schemes.

Note that, if you don't partition a KUDU table, one large partition will be created by default. For example,

```
CREATE TABLE KUDU_TBL_01(COL1 STRING PRIMARY KEY)
 STORED AS KUDU;
+-------------------------+
| summary                 |
+-------------------------+
| Table has been created. |
+-------------------------+
WARNINGS: Unpartitioned Kudu tables are inefficient for large data sizes.
```

To create a KUDU table which uses hash partitioning include the PARTITION BY HASH Statement along with the PARTITIONS Clause followed by the number of desired partitions. For example, let's re-create the DIM_POSTAL_CODE table as a KUDU table. As indicated above, a PRIMARY KEY is essential and, for our example, the POSTAL_CODE column is our best bet:

```
CREATE TABLE DIM_POSTAL_CODE_KUDU(POSTAL_CODE STRING PRIMARY KEY,
                                  CITY STRING,
                                  STATE_CODE STRING,
                                  LATITUDE DOUBLE,
                                  LONGITUDE DOUBLE)
 PARTITION BY HASH (POSTAL_CODE) PARTITIONS 50
 STORED AS KUDU;
```

When using hash partitioning, you must specify the number of partitions you want. Here, we're specifying 50 after the PARTITIONS keyword.

Note that we can replace the column POSTAL_CODE as the hash by, say, the column STATE_CODE, but only if STATE_CODE is also a part of the PRIMARY KEY. Here's how to create a KUDU table with a multi-column PRIMARY KEY:

```
CREATE TABLE DIM_POSTAL_CODE_KUDU(POSTAL_CODE STRING,
                                  STATE_CODE STRING,
                                  CITY STRING,
                                  LATITUDE DOUBLE,
                                  LONGITUDE DOUBLE,
                                  PRIMARY KEY(POSTAL_CODE,STATE_CODE))
 PARTITION BY HASH (STATE_CODE) PARTITIONS 3
 STORED AS KUDU;
```

Take note that the columns as defined in the table must begin with the columns listed in the same order as they appear in the `PRIMARY KEY`. This is why the column `STATE_CODE` was moved up one row in the code above as compared with the previous code.
Now, instead of hash partitioning, range partitioning can be used. For example, let's re-create the table `DIM_POSTAL_CODE` by using range partitioning on the column `STATE_CODE`:

```
CREATE TABLE DIM_POSTAL_CODE_KUDU(POSTAL_CODE STRING,
                                  STATE_CODE STRING,
                                  CITY STRING,
                                  LATITUDE DOUBLE,
                                  LONGITUDE DOUBLE,
                                  PRIMARY KEY(POSTAL_CODE,STATE_CODE))
 PARTITION BY RANGE (STATE_CODE)
 (
  PARTITION VALUE = 'AK',
  PARTITION VALUE = 'AL',
  PARTITION VALUE = 'AR',
  ...skip...
  PARTITION VALUE = 'WI',
  PARTITION VALUE = 'WV',
  PARTITION VALUE = 'WY'
 )
 STORED AS KUDU;
```

Take note that the keyword `VALUE` is used after the keyword `PARTITION` to indicate the relevant value for the partition.

Now, we can combine the two partitioning types hash and range when defining a `KUDU` table. For example, let's hash by the `POSTAL_CODE` column while using range partitioning on the `STATE_CODE`:

```
CREATE TABLE DIM_POSTAL_CODE_KUDU(POSTAL_CODE STRING,
                                  STATE_CODE STRING,
                                  CITY STRING,
                                  LATITUDE DOUBLE,
                                  LONGITUDE DOUBLE,
                                  PRIMARY KEY(POSTAL_CODE,STATE_CODE))
 PARTITION BY
  HASH (POSTAL_CODE) PARTITIONS 100,<-------- WHOA!! COMMA!!
  RANGE (STATE_CODE)
  (
   PARTITION VALUE = 'AK',
   PARTITION VALUE = 'AL',
   PARTITION VALUE = 'AR',
   ...skip...
   PARTITION VALUE = 'WI',
   PARTITION VALUE = 'WV',
   PARTITION VALUE = 'WY'
  )
 STORED AS KUDU;
```

In the syntax above, the keyword `PARTITION BY` appears only once and not once for each requested partition type (`HASH` and `RANGE`).
Now, range partitioning can include a range of values by specifying either the less than symbol (<) or the less than or equal to symbol (<=). For example, let's use range partitioning on the `POSTAL_CODE` column by specifying ranges from `'00000'` to `'09999'`, `'10000'` to `'19999'`, and so on:

```
CREATE TABLE DIM_POSTAL_CODE_KUDU(POSTAL_CODE STRING,
                                  STATE_CODE STRING,
                                  CITY STRING,
                                  LATITUDE DOUBLE,
                                  LONGITUDE DOUBLE,
                                  PRIMARY KEY(POSTAL_CODE))
 PARTITION BY RANGE (POSTAL_CODE)
  (
   PARTITION '00000' <= VALUES < '09999',
   PARTITION '10000' <= VALUES < '19999',
   PARTITION '20000' <= VALUES < '29999',
   PARTITION '30000' <= VALUES < '39999',
   PARTITION '40000' <= VALUES < '49999',
   PARTITION '50000' <= VALUES < '59999',
   PARTITION '60000' <= VALUES < '69999',
   PARTITION '70000' <= VALUES < '79999',
   PARTITION '80000' <= VALUES < '89999',
   PARTITION '90000' <= VALUES < '99999'
  )
 STORED AS KUDU;
```

Naturally, you can use range partitions on numeric values as well.

It's important to note that if one or more of your primary key columns contains a `NULL` value, that row of data is NOT placed in the table and a warning message will be displayed. Always ensure your primary key columns do not contain `NULL` values. For example,

```
CREATE TABLE KUDU_TBL_01(COL1 STRING PRIMARY KEY)
 STORED AS KUDU;

INSERT  INTO KUDU_TBL_01 VALUES('ABC');
INSERT  INTO KUDU_TBL_01 VALUES('DEF');
INSERT  INTO KUDU_TBL_01 VALUES('GHI');
INSERT  INTO KUDU_TBL_01 VALUES(NULL);

WARNINGS: Row with null value violates nullability constraint on table
'impala::prod_schema.KUDU_TBL_01'.

INSERT  INTO KUDU_TBL_01 VALUES('JKL');
```

The same comment is true when using the `INSERT INTO` Statement with a SQL query:

```
CREATE TABLE ADDL_ROWS(COL1 STRING)
 STORED AS PARQUET;

INSERT INTO ADDL_ROWS VALUES('ZZZ');
INSERT INTO ADDL_ROWS VALUES('YYY');
INSERT INTO ADDL_ROWS VALUES('XXX');
INSERT INTO ADDL_ROWS VALUES(NULL);
INSERT INTO ADDL_ROWS VALUES('WWW');

INSERT INTO KUDU_TBL_01
 SELECT COL1
```

```
  FROM ADDL_ROWS;

WARNINGS: Row with null value violates nullability constraint on table
'impala::prod_schema.KUDU_TBL_01'.
```

PART III - Working with the Linux Operating System

Chapter 17 – PuTTY and the Linux Edge Node Server

In *Chapter 3 – Recommended Windows Client Software*, we downloaded, installed and set up PuTTY, the application we'll use to interact with the Linux edge node server. Recall that we also started PuTTY and logged into the Linux edge node server and then promptly typed in the command `exit` to end the session and close PuTTY.

In this chapter, we go over several important features of PuTTY to make your life easier such as cut-and-paste, command history and a variety of very useful options.

Copy/Paste with PuTTY

For the examples below, start your favorite text editor (or Notepad) so we can make use of it later.

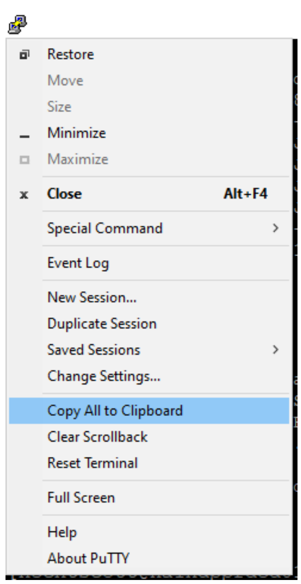
Start PuTTY by double-clicking the Desktop shortcut and then double-clicking the saved session name you gave your Linux edge node server. Once the session comes up, log in using your username and password. At this point, you're at the Linux command prompt and you'll see something like this on the screen (as we've seen before):

```
[smithbob@lnxserver ~]$
```

Now, in the next chapter we'll discuss Linux commands in more detail. Right now, hit the Enter key several times. You should see something thrilling like this:

```
[smithbob@lnxserver ~]$
[smithbob@lnxserver ~]$
[smithbob@lnxserver ~]$
[smithbob@lnxserver ~]$
[smithbob@lnxserver ~]$
[smithbob@lnxserver ~]$
[smithbob@lnxserver ~]$
[smithbob@lnxserver ~]$
```

In order to copy everything appearing on your screen, including the information that has scrolled off, click the application icon at the upper left of PuTTY and click the popup menu item labeled **Copy All to Clipboard**, shown below:



Next, go back to your text editor and hit `CTRL+v`. At this point, all of the text appearing in PuTTY, including those rows that scrolled off, should appear. This is one way to copy from PuTTY to a Windows application.

Now, you can go in the opposite direction as well. In your text editor, type the text `ls -alF`. This is the Linux command which shows the files and folders in the current working directory. Highlight and copy it from your text editor and then move back to PuTTY. Click your right mouse button and this text should appear on the command line in PuTTY. If you just happened to copy the carriage return/line feed as well, the command will automatically execute, so be careful! Also, if you copy multiple lines from your text editor, some or all of those commands will execute automatically. Again, be careful! You don't want to accidentally launch a nuclear strike…oh, or delete a file.

If you don't want to copy all of the text from PuTTY, but only a portion of it, you can highlight the desired text in PuTTY and it'll automatically be copied to the clipboard. For example, using your mouse, click and drag using the left mouse button from one point of the PuTTY window to another. You'll see several lines of text have been highlighted. Go back to your text editor and hit `CTRL+v` to paste in the text. Alternatively, you can copy a rectangular portion by holding down the `ALT` key while clicking and dragging. This option is great if you want to copy the rectangular shaped output from, say, a SQL query.

If you accidentally hit `CTRL+v` while at the Linux command line, you may see strange characters appear on the screen. Just hit the backspace button a few times and they should go away. This'll remind you that you need to click your right mouse button to paste into PuTTY, and not use `CTRL+v`.

Although we show you how to use the finger-mangling `vi` Editor later in the book, you can actually copy an entire SQL program from Windows into the Linux `vi` Editor at the click of the right mouse button. This method just prevents you from having to FTP the file over using FileZilla or WinSCP. For example, on your Windows laptop, open one of your lovely SQL programs, highlight all of the code and click `CTRL+c` to copy it to the clipboard. Next, back in PuTTY, at the command line enter the text `vi test1.sql` to start the `vi` Editor with a filename of `test1.sql`. Next, hit the Enter key:

```
[smithbob@lnxserver ~]$ vi test1.sql
```

At this point, your PuTTY screen will be replaced by a blank screen with a series of tildes (`~`) running down the left side. You're not in Kansas anymore, Dorothy! Instead, you're now in the `vi` Editor. It's not shiny, it's not fancy, it's just `vi`.

When you start the `vi` Editor, you're automatically placed in **Command Mode** (but, don't get a big head, Wing Commander Bob!). Now, click the lowercase letter `i` to enter **Insert Mode**. You'll see the text `-- INSERT --` at the bottom left of the screen as a reminder that you've switched from Command Mode to Insert Mode. Now, to paste in your SQL code, click the right mouse button. If all went well, your code will appear in the editor window.

Next, let's back out of Insert Mode, save the file and quit the `vi` Editor. Hit the Escape button once and the text `-- INSERT --` will disappear. You're now back in Command Mode, Toto! Next, click `SHIFT+:` to bring up the `vi` Editor command prompt at the bottom of the screen. Finally, type in the letters `wq` (`w` means write and `q` means quit) and hit the Enter key. At this point, you're safely back at the Linux command prompt. Huzzah! We discuss the `vi` Editor in *Chapter 19 – Introduction to the vi Editor* in much more detail.

For fun, let's write your SQL code to the screen. At the command prompt, type the command `cat test1.sql` and hit the Enter key. Your entire SQL program should appear on the screen.

Note that occasionally when you paste some code into ImpalaSQL's command line utility `impala-shell`, you'll see strange text on the screen, shown below in bold:

```
[hdpserver:21000] prod_schema> select count(*)
                                >
alter     connect   delete    describe  exit      help      insert    profile   rerun     set       show
src       tip       update    use       version
compute   create    desc      drop      explain   history   load      quit      select    shell     source
summary   unset     upsert    values    with
                                >  from prod_schema.dim_postal_code
                                >
                                > ;
```

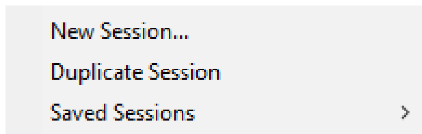
This is due to one or more tabs appearing in your code. You can remove the tabs by using your favorite editor's Regular Expression support. Just replace a tab (`\t`) with a single blank. A similar issue will occur in HiveQL's command line utility `beeline`. It doesn't affect the query, just triggers worldwide OCD.

Recalling Previous Commands

At the Linux command prompt, you can recall a prior command by hitting the up-arrow key. You can continue to do this until you find the desired command. But, see the `history` and `grep` Linux commands in the next chapter before you spend days or weeks clicking the up-arrow button like a dribbling octogenarian.

Connecting to Another PuTTY Session

There are times when you may need to open another session to the Linux edge node server. As shown in the image below, you have several choices from the popup menu, shown below:



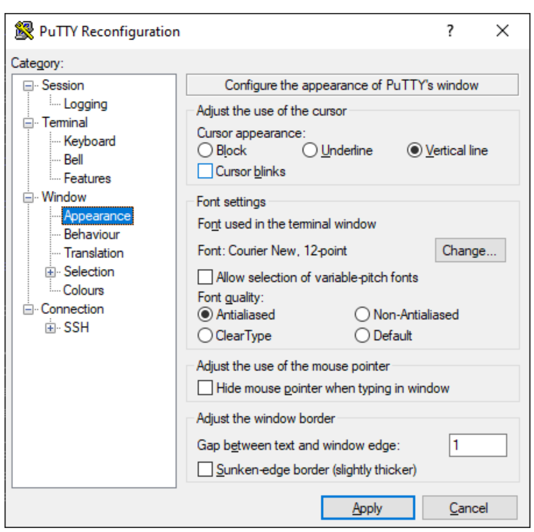
- ☐ **New Session…** will just bring up PuTTY as if you've double-clicked the shortcut on the Desktop.
- ☐ **Duplicate Session** will automatically create another PuTTY session to the server you're currently logged into.
- ☐ **Saved Sessions** allows you to select one of your named sessions.

In all cases, you log in normally and you'll be given a Linux command prompt.

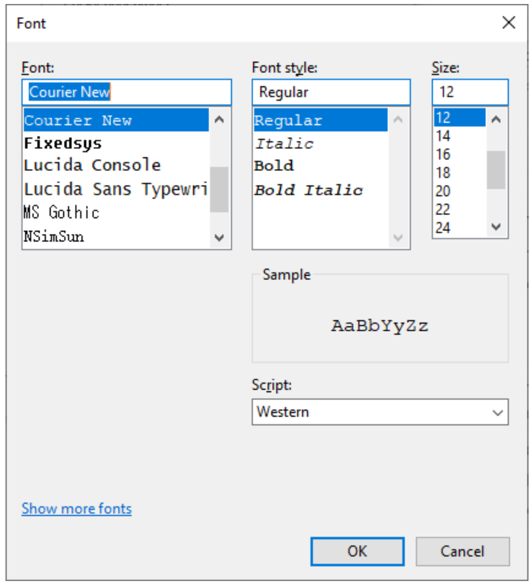
Options Are Good!

Recall when we set up PuTTY, we changed a few options such as turning off the bell and increasing the lines of scroll back. There are several other options you may want to change such as the font size, cursor indicator, and so on.

Now, you can experiment with these changes in the current session by clicking the application icon on the upper left corner of PuTTY and clicking the popup menu item labeled **Change Settings**. Note that if you exit out of PuTTY, your changes won't be saved!! When the PuTTY Reconfiguration dialog appears, click the Appearance node under the Window branch (see below).



On this dialog, you can alter the cursor to appear as a block, an underline or a vertical bar. You also have the option to make the cursor blink, if you check the checkbox to the left of the text **Cursor blinks**. Also, you can change the font family and size by clicking the **Change…** button and selecting from the Font dialog box (shown below).



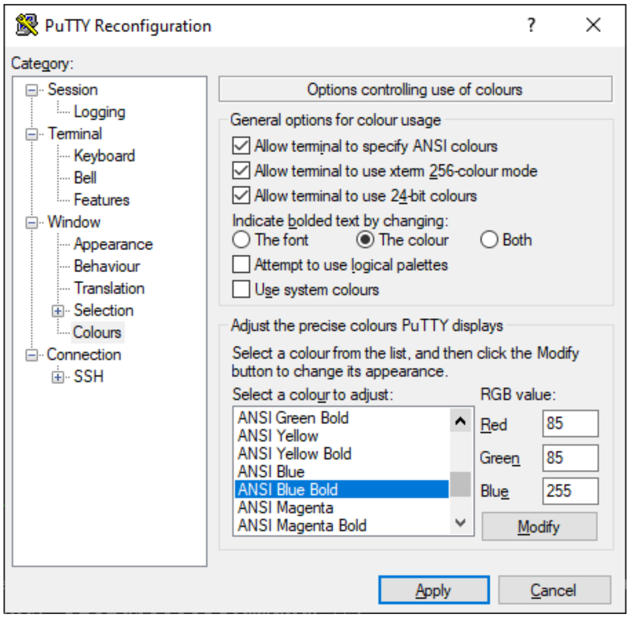
Also, you may want to spend some time looking at the **Font quality** options. The best selection I've found for me is Antialiased, which makes the text easier to read.

Another nice option is to hide the mouse pointer when you begin to type. To turn that on, ensure the checkbox to the left of the text **Hide mouse pointer when typing in window** is checked. You can unhide the mouse by moving the mouse itself. No cheese necessary…eek!!

Next, click Selection under Window. On this dialog, you can alter how the mouse buttons operate as well as how text is copied and pasted. Recall we described how to select a rectangular block of text from the screen when pressing the `ALT` key and dragging the mouse. This option is controlled on this page under **Default selection mode**. If you'd prefer to always use rectangular block mode, click the radio button to the left of the text **Rectangular block**. To use the other mode, hold down the `ALT` key and drag the mouse.

Remaining on the Selection dialog, you can alter how text is copied and pasted in the section labeled **Assign copy/paste actions to clipboards**. I leave the default settings since I'm so used to them, but please check these options out for yourself.

When you begin to edit programs using `vi` in PuTTY, you'll notice that the code has its own colo(u)r scheme. Occasionally, you may not like a particular colo(u)r for, say, the comments. To change the colo(u)r, click on Colours under Selection in the PuTTY Reconfiguration dialog box. In the section labeled **Adjust the precise colours PuTTY displays**, you can alter the colo(u)rs displayed. Below, I've highlighted the ANSI Blue Bold colo(u)r. You can change its RGB value by first clicking on the Modify button and selecting your desired colo(u)r. Clicking Apply will change the colo(u)r immediately. Pretty go(u)o(u)d, huh?



Finally, to save all of your changes, click on the Session node, click on your saved session name and click Save. Your options have been saved and will be used the next time you start PuTTY. Rejoice!

Chapter 18 – Introduction to the Linux Operating System

In this chapter, you'll learn how to work with the Linux operating system from the command line connecting through PuTTY.

Before we start, you should probably be aware that Linux is **case-sensitive**. So, `BOINK` is not the same as `boink`. In Linux, command names are (mostly) in lowercase. You don't have to follow that, it's not a rule, but if one of your programs doesn't work as expected, check the casing before having a conniption fit.

Note that we defer the discussion of submitting/killing jobs, process status and environment variables until *Chapter 20 – Working with Linux Scripts*.

What is Linux?

Linux is an operating system which executes software, maintains files, handles devices, and many, many other things. You most likely use the Microsoft Windows operating system on a daily basis, but you may have heard of other operating systems such as IBM's MVS (used on IBM Mainframes), Digital's VMS (used on Digital VAX), MS-DOS (used on crack), etc. All of these operating systems allow users to analyze data, edit videos, compose music, render 3D graphics, write widely-panned computer books, peruse the *InterWebs* trying to learn what the hell an amp-hour is, etc., all without having to worry about what's going on under the hood.

Unlike Microsoft Windows, Linux is available in different distributions known as *distros*. If you have an old PC/laptop at home, you can probably dust it off, extract any lifeforms living inside it, and bring it back to life by installing Linux on it. Google *Linux distros* and take your pick…there are a lot of them! You may even be able to find the magazine *Linux Format* at your local bookstore (they do exist) which usually comes with a Linux DVD (they do exist) to try out yourself. You can also load a Linux distro on a flash drive and install Linux that way as well.

You can also run Linux on your laptop in a *virtual machine* using software like Oracle's VirtualBox (`www.virtualbox.org`). Download VirtualBox, install it on your laptop, then download a Linux *appliance* to run within the virtual machine. Alternatively, you can download an ISO image of an operating system (such as CentOS, Linux Mint, etc.) and install it in the virtual machine. Voila! Linux on Windows! Nice! (Note: Your operating system's BIOS must support virtualization and it must be enabled, which is not always the case by default.) Recall we discussed some of this in *Chapter 5 – Creating Your Very Own Hadoop Playground* earlier.

There's also a Windows port of Linux called Cygwin. This allows you to execute many Linux commands from either the Windows Command Prompt or from the Cygwin Terminal. Pretty spiffy!! See `www.cygwin.com` for more. I highly recommended installing this on your laptop. We discuss Cygwin in *Appendage #2 – Linux on Windows*.

I WANNA GUI INTERFACE!!

You don't need no stinkin' GUI interface, amigo! Actually, most Linux operating systems do come with a GUI interface (several actually, but that's a story for when you hit puberty) and if you do install Linux on an old PC/laptop, as described in the previous section, that's what you'll see. With that said, running the GUI extensions on your Linux edge node server is not the best use of your server's resources, so it's a black background with white letters…like your father's and your father's father's operating system.

Note that many of your colleagues will balk at learning Linux, won't want to learn Hadoop, and will just want to stick with their SQL programming. And, that's okay, but be aware that it'll happen. In this book, I've tried to include methods which will allow these users to continue to perform their job without the *investment o' time* and *withdrawal o' sanity* it takes to learn all of this stuff. Instead of learning the `vi` Editor, users can create programs on their Windows laptop and then use FileZilla to copy these files over to the Linux server. Subsequent editing can be done by using FileZilla's View/Edit feature which will copy the newly edited and saved file over to the server automatically. When writing SQL, many users can stick to using `STORED AS PARQUET` (which we covered in Part I, *Getting Started*) on their `CREATE TABLE` Statements. With that said, learning all of this stuff – Linux, Hadoop, SQL Analytic functions, Regular Expressions, the `vi` Editor, etc. – will stand you in good stead.

The Linux Directory Structure

The Linux directory structure starts off at the very tippy-top, called the **root directory**, and is indicated by a single forward slash: **/**. Every directory and file exists below the root directory and are named using the forward slash as a delimiter for each subsequent directory. Recall we said your home directory on Linux is `/home/smithbob`. This means there's a folder named `home` under the root directory **/**: **`/home`**. And under the `/home` directory, you'll find your personal directory `smithbob`: **`/home/smithbob`**. If you create a directory in your own account called `python_programs`, the full directory name will be: **`/home/smithbob/python_programs`**.

Now, there are several important directories to be aware of under the root directory (**/**):

- `/tmp` – This directory can be used to store files temporarily. Don't store anything there permanently as it will most likely be deleted. I like to use this folder when creating backups of directories.
- `/home` – This is the directory containing the user directories.
- `/usr/bin` – This directory contains application software.
- `/etc` – This directory contains configuration files used across the entire operating system.

As we demonstrated in *Chapter 1 – Quick Start Guide*, you'll need to refer to either a directory when using `CREATE EXTERNAL TABLE` Statement, or a specific directory or directory/file combination with other applications such as SAS, Python, R, etc. In other words, HDFS uses the same slash-delimited directory/file naming convention as Linux.

Simple Linux Commands

To get started on our Linux adventure, let's learn some basic commands. First, note that there are several ways to refer to a folder (*directory*), or a folder in a folder (*subdirectory*). To avoid this verbal diarrhea, I'll just refer to all that as *directory* and let context indicate if it's *sub* or *not-so-sub*. Now, start PuTTY in the usual way and log in.

At the Linux command line, to determine what directory you're currently in, use the command `pwd`.

`pwd`	**p**rint **w**orking **d**irectory	Displays the current working directory in typical slash format.

```
[smithbob@lnxserver ~]$ pwd
/home/smithbob
```

To see all of the directories and files in the current directory, use the command `ls`.

`ls`	**l**ist **s**tuff	Lists the folders and files in the current directory.

```
[smithbob@lnxserver ~]$ ls
bigmike_output.tsv
```

In order to keeps things organized, you can create your own directories. To create a directory, use the `mkdir` command followed by the name of the directory.

`mkdir`	**ma**ke **dir**ectory	Creates a directory.

```
[smithbob@lnxserver ~]$ mkdir python_programs
```

Recall I mentioned the Linux directory structure is organized with a root at the top indicated by a forward slash. In order to navigate down into a directory, you use the `cd` command followed by the name of the directory.

`cd`	**c**hange **d**irectory	Changes to a different directory.

```
[smithbob@lnxserver ~]$ cd python_programs
[smithbob@lnxserver python_programs]$
```

Take note that your Linux command prompt may display part of the current directory name, as you can see above. This changes automatically as you move around the file system.

If you'd like to go back up one directory, you issue the command **`cd ..`** from the command line. Take note that there's a blank space between `cd` and the two periods. This is just a slight variation on the `cd` command, but instead of providing the name of a directory, you use two periods to indicate *one directory up from the current directory*. Although it may seem silly now, a single period indicates the *current directory*. In summing up,

`.`	*current directory*	Shorthand notation for *current directory*.
`..`	*one directory up from the current directory*	Shorthand notation for *one directory up from current directory (*i.e.*, parent directory)*.
`../..`	*up two directories*	Shorthand notation for *traverse up two directories from current directory*.

```
[smithbob@lnxserver python_programs]$ cd ..
[smithbob@lnxserver ~]$
```

You can also repeat the use of the two dots to indicate you want to traverse up two directories:

```
[smithbob@lnxserver python_programs]$ cd ../..
[smithbob@lnxserver home]$
```

If at any time you just want to get back to your home directory, just issue the command `cd` alone.

```
[smithbob@lnxserver ~]$ cd
[smithbob@lnxserver ~]$ pwd
/home/smithbob
```

You can also go full ninja by using a complete directory path, slashes and all, with the `cd` command:

```
[smithbob@lnxserver ~]$ cd /home/smithbob/python_programs
[smithbob@lnxserver python_programs]$
```

Occasionally, you'll want to create an empty file in a directory up front for use later on. To do this, you use the `touch` command followed by the name of the file.

`touch`	Ye Ol' Midas **touch**	Creates a blank file.

```
[smithbob@lnxserver ~]$ cd python_programs/
[smithbob@lnxserver python_programs]$ touch newfile
[smithbob@lnxserver python_programs]$ ls
newfile
```

Occasionally, you'll have to remove an unwanted file. To do this, you issue the `rm` command followed by the name of the file you want completely and utterly destroyed.

`rm`	**rem**ove	Removes a file.

For example, let's remove the file `newfile`:

```
[smithbob@lnxserver python_programs]$ rm newfile
```

While we're removing files, let's remove a directory. Now, be aware that before you can remove a directory, you must remove the files and directories in it first. Since we just removed a file called `newfile`, our directory `python_programs` is empty. So, let's go back to our home directory and remove the directory `python_programs` using the `rmdir` command.

`rmdir`	**rem**ove **dir**ectory	Removes a directory with joyous abandon.

```
[smithbob@lnxserver python_programs]$ cd
[smithbob@lnxserver ~]$ pwd
/home/smithbob
[smithbob@lnxserver ~]$ rmdir python_programs
[smithbob@lnxserver ~]$
```

To rename a file or directory, use the `mv` command. Let's recreate our `python_programs` directory, create a file named `newfile` and then rename it using `mv` to `newfile1`.

`mv`	**move**	Renames a file/folder or moves a file/folder from one directory to another.

```
[smithbob@lnxserver ~]$ cd
[smithbob@lnxserver ~]$ pwd
/home/smithbob
[smithbob@lnxserver ~]$ mkdir python_programs
[smithbob@lnxserver ~]$ pwd
/home/smithbob
[smithbob@lnxserver ~]$ cd python_programs
[smithbob@lnxserver python_programs]$ pwd
/home/smithbob/python_programs
[smithbob@lnxserver python_programs]$ touch newfile
[smithbob@lnxserver python_programs]$ ls
newfile
[smithbob@lnxserver python_programs]$ mv newfile newfile1
[smithbob@lnxserver python_programs]$ ls
newfile1
[smithbob@lnxserver python_programs]$
```

As you can probably guess from the name of the command, you can also move a file from one place to another on the file system. Let's create a directory under `python_programs` called `_archive` and move the file `newfile1` into it.

```
[smithbob@lnxserver python_programs]$ pwd
/home/smithbob/python_programs
[smithbob@lnxserver python_programs]$ mkdir _archive
[smithbob@lnxserver python_programs]$ ls
_archive  newfile1
[smithbob@lnxserver python_programs]$ mv newfile1 ./_archive
[smithbob@lnxserver python_programs]$ ls
_archive
[smithbob@lnxserver python_programs]$ cd _archive
[smithbob@lnxserver _archive]$ pwd
/home/smithbob/python_programs/_archive
[smithbob@lnxserver _archive]$ ls
newfile1
[smithbob@lnxserver _archive]$ cd ..
[smithbob@lnxserver python_programs]$ pwd
/home/smithbob/python_programs
[smithbob@lnxserver python_programs]$ ls
_archive
[smithbob@lnxserver python_programs]$
```

As you see in the `mv` command above, I made use of the single period (`.`) as an indicator of the *current directory*. It's not strictly necessary to use the period there and you can just specify the `_archive` directory alone, but I feel more comfortable using it than issuing the command `mv newfile1 _archive`. Note that you can also use the `mv` command on directories as well.

You can make a copy of a file using the `cp` command. Let's create a file call `newfile2` in the `python_programs` directory and make a copy of it in the `_archive` folder. Also, while we're at it, let's make a copy of `newfile2` again but this time naming it `newfile2A` in the `_archive` folder.

cp	**cop**y	Creates a copy of a file or folder either in the current or another directory.

```
[smithbob@lnxserver python_programs]$ cd
[smithbob@lnxserver ~]$ pwd
/home/smithbob
[smithbob@lnxserver ~]$ cd python_programs
[smithbob@lnxserver python_programs]$ pwd
/home/smithbob/python_programs
[smithbob@lnxserver python_programs]$ touch newfile2
[smithbob@lnxserver python_programs]$ ls
_archive  newfile2
[smithbob@lnxserver python_programs]$ cp newfile2 ./_archive
[smithbob@lnxserver python_programs]$ ls
_archive  newfile2
[smithbob@lnxserver python_programs]$ cd _archive
[smithbob@lnxserver _archive]$ ls
newfile1  newfile2
[smithbob@lnxserver _archive]$ cd ..
[smithbob@lnxserver python_programs]$ pwd
/home/smithbob/python_programs
[smithbob@lnxserver python_programs]$ ls
_archive  newfile2
[smithbob@lnxserver python_programs]$ cp newfile2 ./_archive/newfile2A
[smithbob@lnxserver python_programs]$ ls
_archive  newfile2
[smithbob@lnxserver python_programs]$ cd _archive
[smithbob@lnxserver _archive]$ ls
newfile1  newfile2  newfile2A
[smithbob@lnxserver _archive]$
```

As mentioned earlier, if you've entered in a very long command, rather than re-entering it again, you can hit the up-arrow key on your keyboard to bring back the last command. You can continue to do this to see your prior commands. Once you find the command you're looking for, you can use your left- and right-arrow keys to move around within the line to edit the command, if necessary. Hit the Enter key to execute the command. If you would like to see all of your past commands, issue the `history` command.

history	**history**	Displays your previously executed commands.

```
[smithbob@lnxserver _archive]$ history
 ...snip...
 1077  pwd
 1078  cd python_programs
 1079  pwd
 1080  touch newfile2
 1081  ls
 1082  cp newfile2 ./_archive
 1083  ls
 1084  cd _archive
 1085  ls
 1086  cd ..
 1087  pwd
 1088  ls
 1089  cp newfile2 ./_archive/newfile2A
 1090  ls
 1091  cd _archive
```

```
 1092  ls
 1093  history
[smithbob@lnxserver _archive]$
```

Although we don't have any files with actual data in it, we can make use of the file `cpuinfo` located in the `/proc` directory which contains information about the CPUs on the server. To see the contents of `cpuinfo`, as well as any file you've created, you use the `cat` command followed by the name of the file.

`cat`	(sound of a cat hacking up a file…that's as good as I could come up with)	Displays the entire (!) contents of a file.

```
[smithbob@lnxserver proc]$ cat /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
stepping        : 2
microcode       : 0x44
cpu MHz         : 3172.558
cache size      : 15360 KB
physical id     : 0
siblings        : 12
core id         : 0
cpu cores       : 6
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 15
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep
bogomips        : 4789.07
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
...snip...
[smithbob@lnxserver proc]$
```

You may not want to see the entire contents of a file, but just a few rows off the top or the bottom of the file. In those instances, you can use the `head` and `tail` commands. Let's see the first ten and last ten rows of `cpuinfo`.

`head`	toupee of the file	Displays the first few rows of a file (default:10).
`tail`	buttock of the file	Displays the last few rows of a file (default:10).

```
[smithbob@lnxserver proc]$ head /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
stepping        : 2
microcode       : 0x44
cpu MHz         : 1200.000
cache size      : 15360 KB
physical id     : 0
```

```
[smithbob@lnxserver proc]$ tail /proc/cpuinfo
fpu_exception   : yes
cpuid level     : 15
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep
bogomips        : 4794.10
clflush size    : 64
cache_alignment : 64
address sizes   : 46 bits physical, 48 bits virtual
power management:
```

Instead of ten rows, which is the default, you can specify any desired number of rows just after the command name. Let's display just two rows instead using `head` and `tail`:

```
[smithbob@lnxserver proc]$ head -2 /proc/cpuinfo
processor       : 0
vendor_id       : GenuineIntel

[smithbob@lnxserver proc]$ tail -2 /proc/cpuinfo
power management:
                                    ← blank line
[smithbob@lnxserver proc]$
```

Note that the file contains a single blank line at the end of it, which is why you see a blank line from the output of `tail`.

You probably noticed that we preceded the number `2` with a dash. This indicates a desired option, or *switch*, for the command. Many Linux commands have several switches allowing you to tailor the functionality of the command to produce your desired result. For example, the `rm` command (**rem**ove files) has the `-i` switch which forces `rm` to ask (`i`=interrogate) you if you really, really, really want to remove the file. For example, let's kill off `newfile2`:

```
[smithbob@lnxserver proc]$ cd
[smithbob@lnxserver ~]$ pwd
/home/smithbob
[smithbob@lnxserver ~]$ cd python_programs
[smithbob@lnxserver python_programs]$ ls
_archive  newfile2
[smithbob@lnxserver python_programs]$ rm -i newfile2
rm: remove regular empty file ‘newfile2’? y
[smithbob@lnxserver python_programs]$ ls
_archive
[smithbob@lnxserver python_programs]$
```

Note that you can answer `y` or `yes` (or variations on capitalization) to the question and `rm` will remove the file. Anything else, the file won't be deleted. By default, the `rm` command just removes the file, no questions asked.

Recall that we used the `cat` command to display the entire contents of a file to the screen. We also used `head` and `tail` to display a certain number of rows from the top or bottom of the file. You can use the `grep` command to search for lines **in a file** and display them to the screen. For example, let's search for the text `model name` in the file `/proc/cpuinfo`.

`grep`	Icelandic for the word *search*?	Search for lines **in a file**.

```
[smithbob@lnxserver python_programs]$ grep "model name" /proc/cpuinfo
model name      : Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
model name      : Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
...snip...
```

Note that you follow the command `grep` with your search criteria in quotes followed by the name of the file you want to search through. Now, if you want to search through your file *ignoring case*, you can use the `-i` switch. Let's search for the word `intel` throughout `/proc/cpuinfo` ignoring the case of the search term:

```
[smithbob@lnxserver python_programs]$ grep -i "intel" /proc/cpuinfo
vendor_id       : GenuineIntel
model name      : Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep intel_ppin
intel_stibp flush_l1d
vendor_id       : GenuineIntel
...snip...
```

Note: When copying commands with switches from applications, such as Microsoft Word, or a variety of browsers, you may run into errors when running these commands caused by an unassuming looking dash, in reality, being an **em dash**. You should change the em dash to a proper dash before issuing the command. Although the following command may look correct…

```
grep –i "intel" /proc/cpuinfo
```

…it contains an *em dash* before the letter `i` and Linux will barf at it.

> *Those who cannot remember to convert a dash to a dash are condemned to repeat em error.*
>
> – Santayana

Words to live by, really.

Finally, it's been a long day and it's time to go home. Type `exit` at the Linux command prompt to exit out of your Linux session and close PuTTY. Now, on to the freeway…*zoooooom*!

More Linux Commands

As mentioned in the previous section, almost every Linux command has a variety of options which you can turn on or off depending on your desired result. These options are controlled by *switches*: single letters and numbers preceded by a dash. Alternatively, some switches take two dashes and a descriptive label which is less cryptic than a single letter (`--quiet` really is better than `-q`).

For example, we used the **ls** command to **l**ist **s**tuff in the previous section. You can follow `ls` by the `-R` switch to display all of the subdirectories and files recursively down. For example, let's use `ls -R` on our `python_programs` folder.

```
[smithbob@lnxserver ~]$ cd
[smithbob@lnxserver ~]$ pwd
/home/smithbob
[smithbob@lnxserver ~]$ ls -R python_programs
python_programs:
_archive

python_programs/_archive:
newfile1  newfile2  newfile2A
[smithbob@lnxserver ~]$
```

As you see in the output above, the contents of the folder `python_programs` is displayed, but also the contents of the `_archive` folder below it is displayed as well. Now, there are three options I generally use instead of just the lone `ls` command: `ls -alF`:

- `-a` – this switch causes all files in the folder to be displayed, including those files beginning with a single period. Files beginning with a single period are called *hidden files* and generally indicate important Linux-specific files such as `.bash_profile`, etc. which you probably shouldn't touch. We talk more about `.bash_profile` further below and we will touch it! So there, big corporate mahoffs! HA!
- `-l` (lowercase letter `L`) – this switch causes the output to be presented in the more detailed long listing format which includes the file size, owner, creation date/time, and more.
- `-F` – this switch causes all executable files to be displayed with an asterisk appended to the name (among other things).

Let's see what `ls -alF` looks like when run on our `python_programs` folder:

```
[smithbob@lnxserver ~]$ ls -alF python_programs
total 16
drwxr-xr-x  3 smithbob hdpbob_users   152 Oct 22 10:33 ./
drwx------ 51 smithbob hdpbob_users 16384 Oct 22 09:40 ../
drwxr-xr-x  2 smithbob hdpbob_users   152 Oct 22 09:57 _archive/
```

And, let's add the recursive switch `-R` as well and run it again:

```
[smithbob@lnxserver ~]$ ls -alFR python_programs
python_programs:
total 16
drwxr-xr-x  3 smithbob hdpbob_users   152 Oct 22 10:33 ./
drwx------ 51 smithbob hdpbob_users 16384 Oct 22 09:40 ../
drwxr-xr-x  2 smithbob hdpbob_users   152 Oct 22 09:57 _archive/

python_programs/_archive:
total 0
drwxr-xr-x 2 smithbob hdpbob_users 152 Oct 22 09:57 ./
drwxr-xr-x 3 smithbob hdpbob_users 152 Oct 22 10:33 ../
-rw-r--r-- 1 smithbob hdpbob_users   0 Oct 22 09:41 newfile1
-rw-r--r-- 1 smithbob hdpbob_users   0 Oct 22 09:56 newfile2
-rw-r--r-- 1 smithbob hdpbob_users   0 Oct 22 09:57 newfile2A
```

I like the `ls -alF` output format so much that I create an alias for it whenever I access a new Linux account. An *alias* is just an alternate name for an entire command, switches and all. As you can imagine, typing in `ls -alF` all of the time can be a right pain in the Tortugas.

`alias`	**alias**	Associates a spritely short name with a long-winded command.

Let's create the alias `lsf` for the command `ls -alF`:

```
[smithbob@lnxserver ~]$ alias lsf='ls -alF'
[smithbob@lnxserver ~]$ lsf python_programs
total 16
drwxr-xr-x  3 smithbob hdpbob_users   152 Oct 22 10:33 ./
drwx------ 51 smithbob hdpbob_users 16384 Oct 22 09:40 ../
drwxr-xr-x  2 smithbob hdpbob_users   152 Oct 22 09:57 _archive/
```

Unfortunately, when you log out of your account, you'll lose the alias. We show you how to add an alias to the `.bash_profile` file after discussing the `vi` Editor.

Recall we talked about the `rm` command and its switch `-i` (which asks if you really want to delete a file). Whenever I access a new account on a Linux server, I create an alias for `rm` and set it to `rm -i`. Effectively, you're hot-wiring `rm` so you'll always be violently **i**nterrogated whenever you attempt to remove a file. Now, when programming Linux scripts, this can cause problems, so `rm` cleverly has the `-f` switch which **f**orces the removal of a file even if the `-i` switch appears.

```
[smithbob@lnxserver ~]$ cd python_programs
[smithbob@lnxserver python_programs]$ cd _archive
[smithbob@lnxserver _archive]$ ls
newfile1  newfile2  newfile2A
[smithbob@lnxserver _archive]$ alias rm='rm -i'
[smithbob@lnxserver _archive]$ rm newfile2A
rm: remove regular empty file 'newfile2A'? y
[smithbob@lnxserver _archive]$ ls
newfile1  newfile2
[smithbob@lnxserver _archive]$ rm -f newfile2
[smithbob@lnxserver _archive]$ ls
newfile1
[smithbob@lnxserver _archive]$
```

If you'd like to know the number of lines in a file, you can use the `wc` command along with its `-l` switch to limit its output to **l**ine count only.

wc	**w**ord **c**ount	Counts the number of characters, words and lines.

```
[smithbob@lnxserver _archive]$ wc /proc/cpuinfo
624 4848 27378 /proc/cpuinfo
[smithbob@lnxserver _archive]$ wc -l /proc/cpuinfo
624 /proc/cpuinfo
```

The first command indicates that there are `624` lines, `4848` words and `27378` characters in the file `/proc/cpuinfo`. The second command limits the output to just lines (`-l`).

As indicated in *Chapter 2 – Hadoop Administrator E-Mail*, when you transfer a text file to the Linux server, it's probably a good idea to run the `dos2unix` command on it to convert any Windows carriage control/line feeds (CRLFs) to the Linux line feed character.

dos2unix	DOS to Unix	Converts Windows CRLFs to Linux Line Feeds.

```
[smithbob@lnxserver _archive]$ dos2unix newfile1
dos2unix: converting file newfile1 to Unix format ...
[smithbob@lnxserver _archive]$
```

Recall that the Linux `grep` command allows you to search through a file for some text. The Linux `find` command, on the other hand, allows you to *search through directories for a file with a specific name*. For example, let's search for the file `newfile1` starting the search from the home directory `/home/smithbob`.

find	**find**s files	Searches for files and folders.

```
[smithbob@lnxserver ~]$ cd
[smithbob@lnxserver ~]$ pwd
/home/smithbob
[smithbob@lnxserver ~]$ find /home/smithbob -name "newfile1"
/home/smithbob/python_programs/_archive/newfile1
```

The Linux `find` command's first argument is the starting directory. Here, we're starting the search at the folder `/home/smithbob`. The `find` command will traverse this folder on down searching for the file. Next, the `-name` switch is followed by the search criteria in quotes. Here, we're searching for `newfile1`. Note that if you want to start searching from the current directory and traverse down, you can specify the period (`.`) as the first argument:

```
[smithbob@lnxserver ~]$ find . -name "newfile1"
./python_programs/_archive/newfile1
```

Notice that a period (`.`) is specified instead of the current directory (`/home/smithbob`, in this case) in the output.

The Linux `echo` command allows you to write some text to the PuTTY screen (also called the *terminal*). While seemingly something you'd only use if you were terribly bored, this function will be used in *Chapter 20 – Working with Linux Scripts* to print some useful information to a log file. For example, let's write the text "Program complete." to the terminal.

echo	**echo**	Writes text to the terminal.

```
[smithbob@lnxserver ~]$ echo "Program complete."
Program complete.
[smithbob@lnxserver ~]$
```

If at any point while logged into the Linux server, you get lost in the space-time continuum, you can issue the Linux `date` command to display the current date and time.

date	**date**	Writes the current date and time to the terminal.

```
[smithbob@lnxserver ~]$ date
Sat Oct 23 13:49:27 EST 2021
[smithbob@lnxserver ~]$
```

Just like other Linux commands, the `date` command takes a series of very useful switches. For example, you can format the output of the `date` command to display only the four-digit year using the `%Y` format:

```
[smithbob@lnxserver jobs]$ date +%Y
2021
[smithbob@lnxserver jobs]$
```

You can include the two-digit month as well by including `%m` as part of the format:

```
[smithbob@lnxserver jobs]$ date +%Y%m
202110
[smithbob@lnxserver jobs]$
```

You can shift the date back and forth by a certain number of, say, months using the `-d` switch. Let's subtract one month from the current month:

```
[smithbob@lnxserver jobs]$ date +%Y%m
202110
[smithbob@lnxserver jobs]$ date -d "-1 month" +%Y%m
202109
[smithbob@lnxserver jobs]$
```

The `date` command, as well as the switches presented above, will be very useful when passing in parameters to ImpalaSQL and HPL/SQL commands, as we'll see later on.

If all of these Linux commands and switches are *gettin' ya' down 'n' causin' your noggin' to zap*, you can just look them up in the Linux manual by using the `man` command followed by the command you're interested in. Let's look up `ls` in the manual.

man	**man**ual	The Lovely Linux Manual.

```
[smithbob@lnxserver _archive]$ man ls
LS(1)                  User  Command                              LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...
```

```
DESCRIPTION
       List  information  about  the  FILEs  (the  current  directory  by  default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print C-style escapes for nongraphic characters
...snip...
Manual page ls(1) line 1 (press h for help or q to quit)
```

Now, `man` takes over your entire screen and can be a bit shocking at first. Although I snipped the entire `man` page for `ls` in the output above, I did leave in what's displayed at the bottom of the screen. Take note that you can hit the letter `q` to exit out of the `man` page. This will get you back to the Linux command prompt and back to safety. To navigate the manual, hit the `f` key to move **f**orward one full screen, hit the `b` key to move **b**ackward one full screen, use the up- and down-arrows to move one line at a time up and down the page.

Note that you can search through the entire Linux manual by using the `-k` switch followed by the words you'd like to search for. For example, let's search for the text `sexy knickers` throughout the Linux manual:

```
[smithbob@lnxserver _archive]$ man -k "sexy knickers"
sexy knickers: nothing appropriate.
```

And quite right, `sexy knickers` isn't appropriate to search for in the Linux manual. Instead, let's try to search for the text `merge` in the Linux manual:

```
[smithbob@lnxserver _archive]$ man -k "merge"
envz_merge (3)       - environment string support
augenrules (8)       - a script that merges component audit rule files
intltool-merge (8)   - merge translated strings into various types of file
intltool-update (8)  - updates PO template file and merge translations with it
merge (1)            - three-way file merge
mergerepo (1)        - Merge multiple repositories together
msgmerge (1)         - merge message catalog and template
paste (1)            - merge lines of files
paste (1p)           - merge corresponding or subsequent lines of files
pdfunite (1)         - Portable Document Format (PDF) page merger
ppdmerge (1)         - merge ppd files
rcsmerge (1)         - merge RCS revisions
sdiff (1)            - side-by-side merge of file differences
sort (1p)            - sort, merge, or sequence check text files
stap-merge (1)       - systemtap per-cpu binary merger
strace-log-merge (1) - merge strace - ff - tt output
tcpslice (8)         - extract pieces of and/or merge together tcpdump files
vgmerge (8)          - Merge volume groups
zipmerge (1)         - merge zip archives
[smithbob@lnxserver _archive]$
```

As you can see, a lot of output is displayed. But, perusing through this list may remind you of a command or can help you find an appropriate command to use. Yeah, there's always the Internet, but seeing how that's just a fad and'll go away soon, best to learn how to use `man`.

More Than One?

In many of the commands shown above, we were focused on a specific file or folder which was indicated in the syntax of the command. In Linux, though, you can use syntax similar to Regular Expressions with many of these commands. Recall that I beat you into submission over Regular Expressions in *Chapter 11 – Regular Expressions*. For example, let's use the `find` command to search for all files that start with the text `newfile` followed by additional text:

```
[smithbob@lnxserver ~]$ find . -name "newfile*"
./python_programs/_archive/newfile1
./python_programs/_archive/newfile2
./python_programs/_archive/newfile3
./python_programs/_archive/newfile4A
./python_programs/_archive/newfile4
```

As you can see, `newfile4A` is part of the output. Instead, let's search for files beginning with the text `newfile`, but ending in only the numbers `1`, `2`, `3` or `4`:

```
[smithbob@lnxserver ~]$ find . -name "newfile[1234]"
./python_programs/_archive/newfile1
./python_programs/_archive/newfile2
./python_programs/_archive/newfile3
./python_programs/_archive/newfile4
```

Naturally, let's use our `lsf` alias to do something similar:

```
[smithbob@lnxserver ~]$ cd
[smithbob@lnxserver ~]$ cd python_programs
[smithbob@lnxserver python_programs]$ cd _archive
[smithbob@lnxserver _archive]$ lsf
total 8
drwxr-xr-x 2 smithbob hdpbob_users 8192 Oct 23 13:15 ./
drwxr-xr-x 3 smithbob hdpbob_users  152 Oct 22 10:33 ../
-rw-r--r-- 1 smithbob hdpbob_users    0 Oct 23 10:24 newfile1
-rw-r--r-- 1 smithbob hdpbob_users    0 Oct 23 13:07 newfile2
-rw-r--r-- 1 smithbob hdpbob_users    0 Oct 23 13:07 newfile3
-rw-r--r-- 1 smithbob hdpbob_users    0 Oct 23 13:08 newfile4
-rw-r--r-- 1 smithbob hdpbob_users    0 Oct 23 13:08 newfile4A
[smithbob@lnxserver _archive]$ lsf newfile[1234]
-rw-r--r-- 1 smithbob hdpbob_users 0 Oct 23 10:24 newfile1
-rw-r--r-- 1 smithbob hdpbob_users 0 Oct 23 13:07 newfile2
-rw-r--r-- 1 smithbob hdpbob_users 0 Oct 23 13:07 newfile3
-rw-r--r-- 1 smithbob hdpbob_users 0 Oct 23 13:08 newfile4
[smithbob@lnxserver _archive]$
```

As you can see, the use of Regular Expressions in Linux commands can definitely enhance your usage of them and brighten your day!

Piping and Redirecting

In an earlier section, we used the Linux `cat` command to write the contents of `/proc/cpuinfo` to the terminal. We also used the `wc` command, along with the `-l` switch, to count the number of **l**ines in that file. But, you can combine the two individual commands by directing the **output of one command** and using it as **input to a**

subsequent command. This is called *piping* or *pipelining* and is indicated on the command line by a vertical bar (called a **pipe**) between the two commands. For example, let's combine two commands together using a pipe:

```
[smithbob@lnxserver _archive]$ cat /proc/cpuinfo | wc -l
624
[smithbob@lnxserver _archive]$
```

In fact, you can perform piping on more than two commands in succession. Below, we're sending the output from the `cat` command as input to the `wc` command in order to count the number of lines (`wc -l`) and then as input into the `wc` command (again) to count the number of words (`wc -w`):

```
[smithbob@lnxserver _archive]$ cat /proc/cpuinfo | wc -l | wc -w
1
[smithbob@lnxserver _archive]$
```

Instead of piping the output of a command into another command, you may want to just place the output in a file. This is called *redirection* and is indicated on the command line with either one greater than symbol (>) or two greater than symbols (>>):

- > - Takes the output from the command on the left and **creates/replaces** the text in the named file on the right. Note that if the file exists, any data contained within it is given the death ray.
- >> - Takes the output from the command on the left and **appends** the text into the named file on the right.

Among other things, these two symbols are great for creating log files for your Linux scripts! Let's take the output from an `echo` command and redirect it into our log file `myprogram_20211015.log`:

```
[smithbob@lnxserver ~]$ echo "Program starts..." > myprogram_20211015.log
[smithbob@lnxserver ~]$ cat myprogram_20211015.log
Program starts...
[smithbob@lnxserver ~]$
```

Let's say that your excellent program has completed successfully, of course! Let's add a message to the log file:

```
[smithbob@lnxserver ~]$ echo "Program ends." >> myprogram_20211015.log
[smithbob@lnxserver ~]$ cat myprogram_20211015.log
Program starts...
Program ends.
[smithbob@lnxserver ~]$
```

Redirecting (REDUX)

In the previous section, we learned how to take the output of one command and use it as input to another command in a process called *piping*. We also saw how to use one greater than symbol to create or overwrite a file, and use two greater than symbols to append to a file in a process called *redirection*. In Linux, commands take input, perform some magic on that input, and possibly produce some output as well as error messages, if any. These three things – input, output, error – are known as *streams* and are associated with special names in Linux which you may see mentioned in Linux `man` pages or other documentation:

- `STDIN` – Standard Input – Indicates the input used by a command (occasionally, `stdin`).
- `STDOUT` – Standard Output – Indicates the output produced by a command (occasionally, `stdout`).
- `STDERR` – Standard Error – Indicates any error messages produced by a command (occasionally, `stderr`).

Each of these three streams is further associated with a unique integer:

- 0 – STDIN
- 1 – STDOUT
- 2 – STDERR

Normally, you won't see any of this while working with Linux commands. Any output and error messages produced by a Linux command will just appear on the terminal intermixed. No message appears screaming the words "`HERE'S #1 (STDOUT)!`" or "`HERE'S #2 (STDERR)!`". But, be aware that some commands do distinguish the two streams and you can control where you want those streams to be directed.

Occasionally I use the `STDERR` value `2` when performing redirection. When using the redirection arrows, only `STDOUT` is captured whereas `STDERR` is printed to the terminal. They are, of course, different streams and should be separated, but, when creating log files, it's probably a good idea to capture any error messages so you can peruse them. To do this, you include the following sequence of symbols at the end of the redirection line: **`2>&1`**. Effectively, this indicates you want the error messages normally sent to the `STDERR` (`2`) stream to be forced into the `STDOUT` (`1`) stream. Here's an example:

```
[smithbob@lnxserver ~]$ python mypgm.py > mypgm.log 2>&1
```

Just to reinforce that this stream stuff isn't an obscure topic, here's part of the Python Linux help manual:

```
[smithbob@lnxserver ~]$ python --help
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
...snip...
-i     : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-u     : unbuffered binary stdout and stderr; also PYTHONUNBUFFERED=x
-      : program read from stdin (default; interactive mode if a tty)
...snip...
```

Note that, occasionally, a single dash (**-**) can be associated with `STDIN`, as you can see above.

When generating an output text file using any one of the variety of languages available to you from the Linux command line such as `impala-shell`, `python`, `R`, etc., you may want to ignore any warning/error messages being sent to `STDOUT` and not accidentally place them in, say, an output text file. One way to do that is to redirect `STDERR` to `/dev/null`. This directory is adorably referred to as the *bit bucket* and anything that enters it is sent to the nether regions:

```
[smithbob@lnxserver ~]$ python mypgm.py > client_output_data.txt 2>/dev/null
```

In the code above, the output from the Python program won't include error messages.

Back Itch? No, Silly! Backtick (`)!

Linux has this wonderful feature allowing you to execute a Linux command behind the scenes when surrounded with backtick marks (``). This is useful for many things, but I use it quite a lot to capture the date/time a program started and ended for my log files. For example,

```
[smithbob@lnxserver ~]$ echo "Program started at `date`" >
                                              myprogram_20211015.log
[smithbob@lnxserver ~]$ cat myprogram_20211015.log
Program started at Sat Oct 23 14:10:59 EDT 2021
[smithbob@lnxserver ~]$
```

As you can see, the results of the `date` command have been placed inside the double-quotes. This feature is very useful when writing Linux scripts, as we'll see in the next chapter.

Cleaning Text Files Using `sed` and `tr`

Since we described piping above, we can make use of two very nice text manipulation utilities `sed` and `tr` to alter text on-the-fly.

`sed`	**s**tream **ed**itor	Used to transform text.
`tr`	**tr**anslate	Transforms characters.

For example, using `/proc/cpuinfo`, let's use `sed` to change the word `Xeon` to `386`:

```
[smithbob@lnxserver ~]$ cat /proc/cpuinfo | sed -e 's/Xeon/386/g'
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 63
model name      : Intel(R) 386(R) CPU E5-2620 v3 @ 2.40GHz
stepping        : 2
...snip...
[smithbob@lnxserver ~]$
```

The `sed` command above is using the output from the `cat` command piped as its input. You can also specify a file as well:

```
[smithbob@lnxserver ~]$ sed -e 's/Xeon/386/g' /proc/cpuinfo
[smithbob@lnxserver ~]$
```

The `-e` switch is followed by a `sed` *script* in quotes. A `sed` script may take the form of `s/search-text/replacement-text/g` where `s` indicates the script start and `g` indicates that the `replacement-text` is to be applied across the entire line of text (`g`=global). Note that `search-text` can be a Regular Expression and the `replacement-text` can make use of backreferences. For example, let's replace the incoming text `E5-` followed by four numbers with `ZZ-` those same four numbers:

```
[smithbob@lnxserver ~]$ sed -e 's/E5-\([0-9][0-9][0-9][0-9]\)/ZZ-\1/g'
                                                              /proc/cpuinfo
[smithbob@lnxserver ~]$
```

Note that you must escape both the left and right parentheses which is why `\(` and `\)` appear, but you don't have to escape the backreference `\1`.

Besides text replacement, you can use `sed` to delete one or more rows based on the `search-text` using the `sed` command `d` (for **d**elete). Let's delete the rows in `/proc/cpuinfo` containing the text `model name`:

```
[smithbob@lnxserver ~]$ cat /proc/cpuinfo | sed -e '/model name/d'
[smithbob@lnxserver ~]$
```

Take note that the `replacement-text` is not needed here, nor is the substitution command `s`.

One feature I've used `sed` for is inserting a header row at the top of a newly generated text file. In this case, I use the `sed` command `i` (for **i**nsert). For example, let's create a file containing three rows of data with three comma-separated values, but no header row:

```
[smithbob@lnxserver ~]$ echo "1,2,3" > file_without_header.csv
[smithbob@lnxserver ~]$ echo "4,5,6" >> file_without_header.csv
[smithbob@lnxserver ~]$ echo "7,8,9" >> file_without_header.csv
[smithbob@lnxserver ~]$ cat file_without_header.csv
1,2,3
4,5,6
7,8,9
```

```
[smithbob@lnxserver ~]$
```

Next, let's use the stream editor `sed` to add a header above line `1`:

```
[smithbob@lnxserver ~]$ cat file_without_header.csv |
                          sed -e '1 i COL1,COL2,COL3' > file_with_header.csv
[smithbob@lnxserver ~]$ cat file_with_header.csv
COL1,COL2,COL3
1,2,3
4,5,6
7,8,9
[smithbob@lnxserver ~]$
```

For the script, we're specifying the desired row number (`1`) followed by the insert command (`i`) and we end with the header text: `COL1,COL2,COL3`.

There have been books written and songs sung about the Linux stream editor, so please peruse your favorite Internet sites when you have a moment.

The translate utility `tr` is used to translate characters from one set of characters to another. For example, let's use `tr` to translate all uppercase letters to lowercase letters:

```
[smithbob@lnxserver ~]$ cat /proc/cpuinfo | tr '[:upper:]' '[:lower:]'
...snip...
vendor_id       : genuineintel
cpu family      : 6
model           : 63
model name      : intel(r) xeon(r) cpu e5-2620 v3 @ 2.40ghz
...snip...
[smithbob@lnxserver ~]$
```

You specify the *from-set* followed by the *to-set*, both in quotes:

```
tr 'from-set' 'to-set'
```

You'll recognize both `[:upper:]` and `[:lower:]` as Character Classes, which we discussed in *Chapter 11 – Regular Expressions*.

You can delete characters with `tr` by using the `-d` switch and leaving off the *to-set*. For example, let's delete all of the uppercase letters from `/proc/cpuinfo`:

```
[smithbob@lnxserver ~]$ cat /proc/cpuinfo | tr -d '[:upper:]'
vendor_id       : enuinentel
cpu family      : 6
model           : 63
model name      : ntel() eon()  5-2620 v3 @ 2.40z
[smithbob@lnxserver ~]$
```

Pulling from the InterWebs Using `wget`

The Linux `wget` command allows you to pull data directly from the Internet into a file on the Linux filesystem. Two important switches are the `-q` switch and the `-O` switch. You follow the `-q` switch with the quoted URL you want to pull down. The `-O` switch is followed by the name you want to give the file on the Linux filesystem.

`wget`	**web get**-at-cha-boi	Pulls data off the Internet.

For example, let's pull the world population data directly from the World Bank using their API (they won't mind, they got money). In this case, the format I've chosen is the `csv` format, but be aware that the file actually comes down as a zipped file containing multiple CSV files. First, let's grab the data (shown on multiple lines for clarity):

```
[smithbob@lnxserver ~]$ wget
  -q "https://api.worldbank.org/v2/en/indicator/SP.POP.TOTL?downloadformat=csv"
  -O /home/smithbob/poptotal.zip
```

Although we talk about compressed files further below, let's decompress this file using the Linux `unzip` command:

```
[smithbob@lnxserver ~]$ unzip poptotal.zip
Archive:  poptotal.zip
  inflating: Metadata_Indicator_API_SP.POP.TOTL_DS2_en_csv_v2_3052518.csv
  inflating: API_SP.POP.TOTL_DS2_en_csv_v2_3052518.csv
  inflating: Metadata_Country_API_SP.POP.TOTL_DS2_en_csv_v2_3052518.csv
```

The `unzip` command decompresses the file which, in actuality, contains three individual CSV files: the population data by country, metadata about the indicator, and metadata about the countries. If you've used WinZip before, and I suspect you have, you'll get the idea.

Sending an E-Mail Using `mail`

If you've gone through the trouble to create a script to run your queries (we talk about Linux scripts in *Chapter 20 – Working with Linux Scripts* and capture any output to a log file, you probably want to e-mail yourself that log file. There are several Linux commands to sendmail…er…send mail, and the one I use is called, unceremoniously, `mail`.

`mail`	send an e-**mail**	Sends an e-mail.
`sendmail`	**send** an e-**mail**	Sends an e-mail.

After creating your log file, you can mail it to yourself using the following command:

```
[smithbob@lnxserver ~]$ cat log.log | mail -s "Subject Line"
                                                   smithbob@company.com
```

Naturally, replace the file `log.log` with your log file, change the subject line to something more appropriate and update the e-mail address. The log file `log.log` will appear as the *body of the e-mail*. To send to multiple e-mail addresses, just repeat the command above changing the destination e-mail address.

You can send an attachment using the `-a` switch followed by the name of the file. For example, let's send `us_state_mapping.csv` as an attachment to our e-mail above:

```
[smithbob@lnxserver ~]$ cat log.log | mail -s "Subject Line"
                                          -a us_state_mapping.csv
                                          smithbob@company.com
```

Note that there may be more e-mail utilities available on your Linux server. You may want to ask your Linux Administrator for a recommendation…by e-mail, of course! AH HA HA HA! See what I did there!?!

Forcing Your Files to Lose Some Weight (`zip`/`gzip`)

As we saw earlier, the World Bank population file comes down as a compressed file. We used the command `unzip` to decompress it to access our three CSV files. There are several compression types available on Linux and we discuss them in this section.

For the most part, you'll be using the compression utility `gzip`. When you see a file with the extension `.gz`, it's been compressed using `gzip`. You can decompress it using either `gzip` with the `-d` switch or the `gunzip` command.

`gzip`	**G**NU **zip**	Compresses files in gzip format.
`gzip -d`	**G**NU **zip**	Decompresses files in gzip format.
`gunzip`	**G**NU **zip**	Decompresses files in gzip format.

Note that the Windows applications WinZip, PKZip, 7-Zip, etc. recognize both `.gz` and `.zip` files. Windows File Explorer natively recognizes `.zip` files, but not `.gz` files (at time of publication…about 4:30 in the afternoon). For example, let's compress the World Bank population data (the `.csv` file) using `gzip`:

```
[smithbob@lnxserver ~]$ lsf *.csv
-rw-r--r-- 1 smithbob hdpbob_users 182389 Oct 11 11:07
                                    API_SP.POP.TOTL_DS2_en_csv_v2_3052518.csv
[smithbob@lnxserver ~]$ gzip API_SP.POP.TOTL_DS2_en_csv_v2_3052518.csv
[smithbob@lnxserver ~]$ lsf *.gz
-rw-r--r-- 1 smithbob hdpbob_users 74475 Oct 11 11:07
                                 API_SP.POP.TOTL_DS2_en_csv_v2_3052518.csv.gz
[smithbob@lnxserver ~]$
```

As you see, the compressed file is `74,475` bytes whereas the original file is `182,389` bytes. Now, you can control the speed of compression by specifying one of the switches ranging from `-1` (fastest compression) to `-9` (slowest compression). The default compression is `-6` (meh compression). The slowest compression (`-9`) should yield a smaller compressed file size as compared to the fastest compression (`-1`). For example, let's really trash compact the file down:

```
[smithbob@lnxserver ~]$ gzip -9 API_SP.POP.TOTL_DS2_en_csv_v2_3052518.csv
[smithbob@lnxserver ~]$ lsf *.gz
-rw-r--r-- 1 smithbob hdpbob_users 73619 Oct 11 11:07
                                 API_SP.POP.TOTL_DS2_en_csv_v2_3052518.csv.gz
[smithbob@lnxserver ~]$
```

As you see, the file compressed down to `40.36%` of its original size with the `-9` switch as compared to `40.83%` with the default `-6` switch. With the `-1` switch it's `44.74%`. Nothing to write home about, but you wouldn't write home about this anyway.

Next, let's decompress the file:

```
[smithbob@lnxserver ~]$ gzip -d API_SP.POP.TOTL_DS2_en_csv_v2_3052518.csv.gz
[smithbob@lnxserver ~]$
```

As indicated above, you can use `gunzip` instead of `gzip -d`.

Now, given multiple files, `gzip` compresses **each one individually** and adds `.gz` to the extension of each file. On the other hand, the Linux compression utility `zip` will compress several files into one large zip file, as we saw with the World Bank population data. For example, let's compress the three CSV population files from the World Bank as well as our file `us_state_mapping.csv` from *Chapter 1 – Quick Start Guide* into one zipped file called `all.zip`:

`zip`	**zip**	Compresses files in zip format.
`unzip`	**unzip**	Decompresses files in zip format.

```
[smithbob@lnxserver ~]$ zip all.zip *.csv
 adding: API_SP.POP.TOTL_DS2_en_csv_v2_3052518.csv (deflated 59%)
 adding: Metadata_Country_API_SP.POP.TOTL_DS2_en_csv_v2_3052518.csv (deflated 79%)
 adding: Metadata_Indicator_API_SP.POP.TOTL_DS2_en_csv_v2_3052518.csv
                                                                  (deflated 41%)
 adding: us_state_mapping.csv (deflated 45%)
```

Note that you can use Regular Expressions with both `gzip` and `zip`, as shown above. Since we saw how to use `unzip` in an earlier section, we won't repeat it here.

Another compression type is the `.tar` file, or **t**ape **ar**chive, which is created using the Linux command `tar`. As the name suggests, it can be used to control tape backup devices such as the Digital Linear Tape (DLT) or Super Digital Linear Tape (SDLT) devices (not gone, but forgotten). Although `tar` can control tape backup devices, it can also create a single file containing an entire directory structure (along with files) into one large file with an extension of `.tar`.

Occasionally, you'll see the extensions `.tgz` and `.tar.gz`. These extensions indicate that the `.tar` file has been compressed using `gzip`. Note that the `tar` command has a switch you can use to decompress a `gzip` compressed `.tar` file, so you don't have to run `gunzip` on it.

`tar`	**t**ape **ar**chive	Creates an archive file on disk.

Let's create a compressed archive file called `bu_pp.tgz` from the contents of our folder `/home/smithbob/python_programs`. We'll create this file in the `/tmp` directory and then move it back to our `/home/smithbob` directory.

```
[smithbob@lnxserver ~]$ cd /tmp
[smithbob@lnxserver tmp]$ tar -zvcf bu_pp.tgz /home/smithbob/python_programs
/home/smithbob/python_programs/
/home/smithbob/python_programs/myprogram_20211015.log
/home/smithbob/python_programs/_archive/
/home/smithbob/python_programs/_archive/zzz
/home/smithbob/python_programs/_archive/newfile1
/home/smithbob/python_programs/_archive/newfile2
/home/smithbob/python_programs/_archive/newfile3
/home/smithbob/python_programs/_archive/newfile4A
/home/smithbob/python_programs/_archive/newfile4
[smithbob@lnxserver tmp]$ lsf *.tgz
-rw-r--r-- 1 smithbob hdpbob_users 375 Oct 25 09:42 bu_pp.tgz
[smithbob@lnxserver tmp]$ mv bu_pp.tgz /home/smithbob
[smithbob@lnxserver tmp]$ cd
[smithbob@lnxserver ~]$ pwd
/home/smithbob
[smithbob@lnxserver ~]$ lsf *.tgz
-rw-r--r-- 1 smithbob hdpbob_users 375 Oct 25 09:42 bu_pp.tgz
[smithbob@lnxserver ~]$
```

The `tar` command above starts with several switches, followed by the desired name of the compressed archive file, followed by the directory whose contents we want to archive. The switches above are:

- `-z` – this switch compresses the archive file using g**z**ip.
- `-v` – this switch will display the directories and files being placed into the archive file. (`v`=verbose)
- `-c` – this switch creates a new archive file (`c`=create)
- `-f` – this switch picks up the name of the archive file (`bu_pp.tgz`, in our case). **Note that this switch must be the last switch since it expects the name of the archive file to follow directly.**

Note that occasionally you'll receive the message **file changed as we read it** from `tar` indicating that a change has occurred in one or more files during the archiving process. This could be caused by a running program making

changes to files, the operating system making changes, gamma rays, etc. To work around this, just specify the switch `–warning=no-file-changed`.

To list (but not unarchive) the contents of the archive file to the screen, you can use the `-t` switch along with the `-f` switch followed by the name of the archive file:

```
[smithbob@lnxserver ~]$ tar -tf bu_pp.tgz
home/smithbob/python_programs/
home/smithbob/python_programs/myprogram_20211015.log
home/smithbob/python_programs/_archive/
home/smithbob/python_programs/_archive/zzz
home/smithbob/python_programs/_archive/newfile1
home/smithbob/python_programs/_archive/newfile2
home/smithbob/python_programs/_archive/newfile3
home/smithbob/python_programs/_archive/newfile4A
home/smithbob/python_programs/_archive/newfile4
[smithbob@lnxserver ~]$
```

To unarchive the file, you specify the switches `-v` and `-f` as before, add the switch `–x` and remove the `-z` switch.

- `-x` – this switch extracts the contents of the archive file to disk.

Because I don't want to take the chance of overwriting any of my important files/directories, I tend to create a temporary directory in which to safely unarchive. Below, I create a `tmp` directory under `/home/smithbob` and move the archive file `bu_pp.tgz` there. Take note that I'm making use of both `..` (parent directory) and `.` (current directory) on the `mv` command below because I'm a bad mamma-jamma!

```
[smithbob@lnxserver ~]$ cd
[smithbob@lnxserver ~]$ mkdir tmp
[smithbob@lnxserver ~]$ cd tmp
[smithbob@lnxserver tmp]$ pwd
/home/smithbob/tmp
[smithbob@lnxserver tmp]$ mv ../bu_pp.tgz .
[smithbob@lnxserver tmp]$ ls
bu_pp.tgz
[smithbob@lnxserver tmp]$ tar -xvf bu_pp.tgz
home/smithbob/python_programs/
home/smithbob/python_programs/myprogram_20211015.log
home/smithbob/python_programs/_archive/
home/smithbob/python_programs/_archive/zzz
home/smithbob/python_programs/_archive/newfile1
home/smithbob/python_programs/_archive/newfile2
home/smithbob/python_programs/_archive/newfile3
home/smithbob/python_programs/_archive/newfile4A
home/smithbob/python_programs/_archive/newfile4
[smithbob@lnxserver tmp]$ ls -alFR
.:
total 24
drwxr-xr-x  3 smithbob hdpbob_users   152 Oct 25 10:03 ./
drwx------ 52 smithbob hdpbob_users 16384 Oct 25 10:02 ../
-rw-r--r--  1 smithbob hdpbob_users   375 Oct 25 09:42 bu_pp.tgz
drwxr-xr-x  3 smithbob hdpbob_users   152 Oct 25 10:03 home/

./home:
total 0
drwxr-xr-x 3 smithbob hdpbob_users 152 Oct 25 10:03 ./
drwxr-xr-x 3 smithbob hdpbob_users 152 Oct 25 10:03 ../
drwxr-xr-x 3 smithbob hdpbob_users 152 Oct 25 10:03 smithbob/
```

```
./home/smithbob:
total 0
drwxr-xr-x 3 smithbob hdpbob_users 152 Oct 25 10:03 ./
drwxr-xr-x 3 smithbob hdpbob_users 152 Oct 25 10:03 ../
drwxr-xr-x 3 smithbob hdpbob_users 152 Oct 24 14:13 python_programs/

./home/smithbob/python_programs:
total 16
drwxr-xr-x 3 smithbob hdpbob_users  152 Oct 24 14:13 ./
drwxr-xr-x 3 smithbob hdpbob_users  152 Oct 25 10:03 ../
drwxr-xr-x 2 smithbob hdpbob_users 8192 Oct 23 13:37 _archive/
-rw-r--r-- 1 smithbob hdpbob_users   18 Oct 23 14:06 myprogram_20211015.log

./home/smithbob/python_programs/_archive:
total 16
drwxr-xr-x 2 smithbob hdpbob_users 8192 Oct 23 13:37 ./
drwxr-xr-x 3 smithbob hdpbob_users  152 Oct 24 14:13 ../
-rw-r--r-- 1 smithbob hdpbob_users    0 Oct 23 10:24 newfile1
-rw-r--r-- 1 smithbob hdpbob_users    0 Oct 23 13:07 newfile2
-rw-r--r-- 1 smithbob hdpbob_users    0 Oct 23 13:07 newfile3
-rw-r--r-- 1 smithbob hdpbob_users    0 Oct 23 13:08 newfile4
-rw-r--r-- 1 smithbob hdpbob_users    0 Oct 23 13:08 newfile4A
-rw-r--r-- 1 smithbob hdpbob_users    4 Oct 23 13:37 zzz
[smithbob@lnxserver tmp]$
```

Directory and File Permissions and `chmod`

Recall in the discussion above, we used a variation of the `ls` command with three switches: `ls -alF`. In the output from that command, reproduced in part below, you'll notice a series of letters and dashes in the first column (shown emboldened below):

```
./home/smithbob/python_programs:
total 16
drwxr-xr-x 3 smithbob hdpbob_users  152 Oct 24 14:13 ./
drwxr-xr-x 3 smithbob hdpbob_users  152 Oct 25 10:03 ../
drwxr-xr-x 2 smithbob hdpbob_users 8192 Oct 23 13:37 _archive/
-rw-r--r-- 1 smithbob hdpbob_users   18 Oct 23 14:06 myprogram_20211015.log
```

This string indicates file and directory permissions and is made up of the following:

- ☐ The first position indicates if the row indicates a directory or a file. If the first character is "`d`", then the row indicates a directory. If the first character is a dash "`-`", then the row indicates a file. As you see above, the directory `_archive` is shown with the letter "`d`", and the file `myprogram_20211015.log` is shown with a dash "`-`".
- ☐ The next nine positions are broken up into three sets of three characters indicating directory or file permissions for the **user**, the **group** and the **world**:
    - The first set of three characters pertains to the **user**'s permissions. These are permissions which the user has on directories or files, whether they're your own or not. When you create a file in your own account, it makes sense that you have control over that file, but your colleagues may not be able to access that file or directory.
    - The second set of three characters pertains to the **group**'s permissions. Each user on your Linux edge node server should be placed in a **group**. For example, you and your department's employees may be placed in a group called, say, `datasci` and it's the second set of three characters which controls access to the files and directories for the group. With that said, as a user, you still have control over your own directories and files indicated by the first set of three characters, and these can be changed, if you so desire, to prevent anyone in the group from accessing your directories and files.

  - The third set of three characters pertains to the **world**'s permissions and indicates if you want to allow anyone on the Linux edge node server to access your directories and files. Again, you have control over this.
- ☐ As indicated above, each of the three sets of three characters indicates user, group and world permissions. Within each set, the characters indicate the following:
  - Position #1 – **read** permission. This controls whether the user, group or world can read your file.
  - Position #2 – **write** permission. This controls whether the user, group or world can modify your file.
  - Position #3 – **execute** permission. This controls whether the user, group or world can execute your file (assuming it's a script or program of some sort).

For example, given the string **`-rwxrw-r-x`**, let's break it down:

1. The first position is a dash indicating the row is a file.
2. The first set of three characters pertains to the **user** and is broken up into the following permissions:
   i. **`r`** – The user has **r**ead permission on the file.
   ii. **`w`** – The user has **w**rite permission on the file.
   iii. **`x`** – The user has e**x**ecute permission on the file.
3. The second set of three characters pertains to the **group** and is broken up into the following permissions:
   i. **`r`** – The group has **r**ead permission on the file.
   ii. **`w`** – The group has **w**rite permission on the file.
   iii. **`-`** – The group does **not** have execute permission on the file.
4. The third set of three characters pertains to the **world** and is broken up into the following permissions:
   i. **`r`** – The world has **r**ead permission on the file.
   ii. **`-`** – The world does **not** have write permission on the file.
   iii. **`x`** – The world has e**x**ecute permission on the file.

Now, you can modify the permissions using the Linux command `chmod` along with its switches. Although this all may seem esoteric, you'll make use of the `chmod` command when you create your own Linux scripts. For example, let's create a new file (using the `touch` command) called `myfile`:

```
[smithbob@lnxserver ~]$ cd python_programs/
[smithbob@lnxserver python_programs]$ touch myfile
[smithbob@lnxserver python_programs]$ ls -alF
total 12
drwxr-xr-x  2 smithbob hdpbob_users  4096 Nov  7 13:17 ./
drwxr-xr-x 45 smithbob hdpbob_users  4096 Nov  7 13:15 ../
-rw-r--r--  1 smithbob hdpbob_users     0 Nov  7 13:17 myfile
[smithbob@lnxserver python_programs]$
```

Notice that `myfile` has the following permissions: `-rw-r--r--`. Let's make the file executable using `chmod`:

```
[smithbob@lnxserver python_programs]$ chmod +x myfile
```

Now the file has the following permissions on it: `-rwxr-xr-x`. As you see, the user, group and world now have execute permission on your file. This is probably not what you want, but be aware that this form of the `chmod` command exists.

`chmod +r` *`file`*	**r**ead permission	Grant read permission on *`file`* for user, group and world.
`chmod +w` *`file`*	**w**rite permission	Grant write permission on *`file`* for user, group and world.
`chmod +x` *`file`*	e**x**ecute permission	Grant execute permission on *`file`* for user, group and world.

In the form of the `chmod` above, you can specify a dash (`-`) instead of a plus (`+`) to indicate that that permission should be revoked: `chmod -x myfile`.

Another form of the `chmod` command allows you to specify the letters `u` (user), `g` (group), `o` (other/world) and `a` (all three) followed by an equal sign followed by the permission(s) you want to grant: `r` (read), `w` (write) and `x` (execute). For example, let's grant the **user** read, write and execute permissions; the **group**, read and execute permissions; and the **world**, read permission only:

```
[smithbob@lnxserver python_programs]$ chmod u=rwx,g=rx,o=r myfile
[smithbob@lnxserver python_programs]$ lsf
total 12
drwxr-xr-x  2 smithbob hdpbob_users 4096 Nov  7 13:17 ./
drwxr-xr-x 45 smithbob smithbob   4096 Nov  7 13:15 ../
-rwxr-xr--  1 smithbob hdpbob_users    0 Nov  7 13:17 myfile*
[smithbob@lnxserver python_programs]$
```

You can also dispense with the equal sign and specify a dash (-) or a plus sign (+) to revoke or grant one or more permissions. For example, instead of the command shown above, let's use this one instead:

```
[smithbob@lnxserver python_programs]$ chmod u+rwx,g+rx,o+r myfile
```

The results are the same as above.

The final form of `chmod` expects you to specify permissions using numeric values rather than letters. These numeric values can be computed by the following formula:

$$permission\ value = r2^2 + w2^1 + x2^0 = 4r + 2w + x$$

where r, w and x play the role of indicators in this case:

- ☐ If you want the file to have **read** permission, set $r = 1$; otherwise, 0.
- ☐ If you want the file to have **write** permission, set $w = 1$; otherwise, 0.
- ☐ If you want the file to have **execute** permission, set $x = 1$; otherwise, 0.

Note that you'll have to compute the permission value as a concatenation of the permission values for the user, the group and the world.

For example, using our example above, let's give the **user** read, write and execute permissions; the **group** read and execute permissions; and the **world** read permission only:

$$\begin{aligned} user\ permission\ value &= 1\times 2^2 + 1\times 2^1 + 1\times 2^0 = 4\times 1 + 2\times 1 + 1\times 1 = 7 \\ group\ permission\ value &= 1\times 2^2 + 0\times 2^1 + 1\times 2^0 = 4\times 1 + 2\times 0 + 1\times 1 = 5 \\ world\ permission\ value &= 1\times 2^2 + 0\times 2^1 + 0\times 2^0 = 4\times 1 + 2\times 0 + 1\times 0 = 4 \end{aligned}$$

The corresponding `chmod` syntax is:

```
                                           111 101 100
[smithbob@lnxserver python_programs]$ chmod 754 myfile
```

One nice use of `chmod` is to set file permissions so that only you can read the file, such as a file containing passwords or other sensitive material. For example, the code below makes the file `password_file` read only for the user…that would be you. (Although naming the file `password_file` is pushing it a bit!)

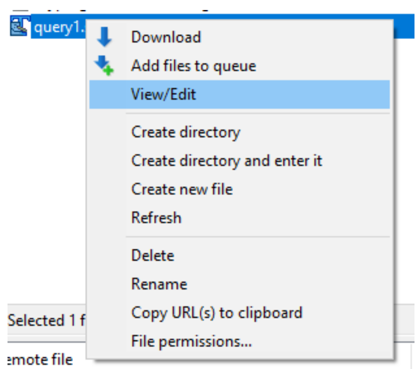
```
                                           100 000 000
[smithbob@lnxserver python_programs]$ chmod 400 password_file
```

Chapter 19 – Introduction to the `vi` Editor

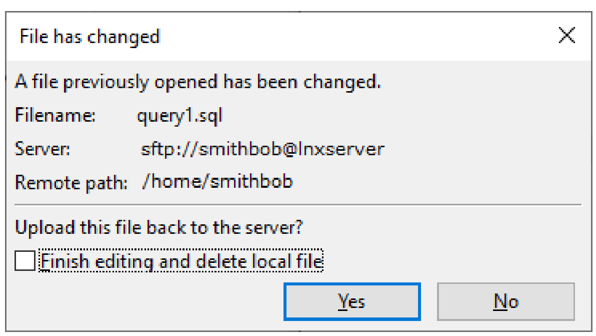
In this chapter, you'll learn how to use the `vi` Editor. But, before we do that, let's take a look at FileZilla's View/Edit feature. First, create a text file called `query1.sql` on your laptop containing the following fab SQL code:

```
select state_code,state_name
 from prod_schema.dim_us_state_mapping
 order by state_code;
```

Next, use FileZilla to transfer this file from your laptop to the Linux edge node server into the `/home/smithbob` directory. You can drag from the left side (representing your **Windows laptop**) and drop onto the right side (representing the **Linux server**). Next, right-click on the file `query1.sql` **from the right side** (the Linux side) and click the View/Edit popup menu item:



FileZilla will transfer the file from the Linux server back to your laptop, but into a temporary file location. Edit the file by changing the `ORDER BY` Clause to sort by `state_name`, save and then close the file. FileZilla will display a popup dialog box with the text **File has changed** on its title bar (see below). Click the Yes button to upload the file back to the Linux server.



Ensure the checkbox to the left of the text **Finish editing and delete local file** is checked, assuming you've finished editing your file. Now, using PuTTY, log into the server and `cat` the file to see if the changes have actually been made. Of course they have…you ain't no penny-ante operator! This is one way to interact with a file on the Linux server without getting your hands all dirty with sticky `vi` Editor spew.

Now, let's trudge into the aforementioned `vi` Editor spew. As mentioned before, the `vi` Editor is the lowest common denominator among all Linux and Unix operating systems. Rough translation: It's probably a good idea to learn it. Yes, the `vi` Editor is finger-twistingly violent, but it gets the job done. (There are other editors, like Emacs, but you can look into that in your vast amounts of free time.)

Let's create a new file using the `vi` Editor. Log into the Linux edge node server into your account `/home/smithbob`. At the Linux command line, type in the following and hit the Enter key:

```
[smithbob@lnxserver ~]$ vi bobsmith.txt
```

At this point, your screen is taken over by the `vi` Editor's display:

```
~
~
~
~
~
~
...tilde snip...
~
~
~
~
~
~
"bobsmith.txt" [New File]
```

The tildes indicate that nothing is on the line. At the bottom of the screen, you'll see the name of the file and an indication that the file is new.

Now, there are two modes in the `vi` Editor: **Insert Mode** and **Command Mode**. When you first start the `vi` Editor, you're automatically placed in Command Mode, which allows you to copy, paste, delete, shift, etc. the lines of text appearing in the file. Insert Mode allows you to enter in text just like your favorite text editor (or Notepad). Let's enter into Insert Mode. Click the lowercase letter `i` to enter into Insert Mode. At the bottom of the screen, the name of the file and the `[New File]` indicator disappear and are replaced with the text **`-- INSERT --`**. At this point, you can just type normally. Enter the following text: `The quick brown fox jumped over the lazy antelope.`, but do not hit the Enter key.

Now, to exit Insert Mode, hit the escape key (ESC) at the upper-left hand corner of your keyboard. At the bottom of the screen, the Insert Mode indicator **`-- INSERT --`** goes buh-bye.

`i`	**i**nsert	Enters Insert Mode.
`ESC`	**ESC**ape key	Exits Insert Mode and enters Command Mode.

Note that you'll eventually find yourself mindlessly hitting the Escape key over and over again. This happens to all users of the `vi` Editor at some point. Don't fear it!

At this point, your cursor should be at the end of the line, indicated by an ominous dark rectangle:

```
The quick brown fox jumped over the lazy antelope.█
~
~
```

Since you're no longer in Insert Mode, but in Command Mode, let's move the cursor to the beginning of the line. Hitting the number zero (`0`) will move your cursor to the beginning of the line. Take note that your cursor has jumped to the beginning of the line. To move the cursor back to the end of the line, hit the dollar sign (`$`).

`0`	Number Zero (**`0`**`xxxxx`)	Moves cursor to the beginning of the line.
`$`	Dollar Sign (`xxxxx`**`$`**)	Moves cursor to the end of the line.

Note that while in Command Mode, you can use the arrow keys to move around left or right one character at a time and up or down one line at a time. Note that you can use the arrow keys in Insert Mode as well:

←	Left Arrow	Moves cursor left one character at a time.
→	Right Arrow	Moves cursor right one character at a time.
↑	Up Arrow	Moves cursor up one line at a time.
↓	Down Arrow	Moves cursor down one line at a time.

In Command Mode, to move right a word at a time, hit the lowercase `w` key, and to move left a word at a time, hit the lowercase `b` key.

`w`	for**w**ord	Moves cursor to the beginning of the next word.
`b`	**b**ackword	Moves cursor to the beginning of the previous word.

To enter additional text at the beginning of a line, just hit `0` to get back to the beginning of the line and then hit `i` to enter Insert Mode. Don't forget to hit the ESCape key to leave Insert Mode and enter Command Mode.

To insert additional text somewhere between the start and end of the line, move your cursor to the desired spot and hit `i` to enter Insert Mode. Note that hitting `i` will put you in Insert Mode *one character before your cursor*. To insert additional text *one character after your cursor*, hit the lowercase letter `a`. To append text starting at the end of the line, hit the capital letter `A` (`SHIFT+A`).

`a`	**a**ppend (`++`**a**`+++`)	Enter Insert Mode one character before your cursor.
`A`	**A**ppend (`+++++`**A**)	Enter Insert Mode at the end of the line.

Next, to delete characters one at a time at your cursor's location, hit the lowercase letter `x`. This will shift the line from the right over to the left each time you hit `x`, but your cursor will stay put until there's no more text to the right of the cursor. To delete characters from your cursor's location to the left, hit the uppercase letter `X` (`SHIFT+X`). Your cursor will shift left each time you hit `X`.

`x`	e**x**punge	Delete characters after your cursor.
`X`	E**X**PUNGE	Delete characters before your cursor.

To open up a new line below the current line (where the cursor is located), hit the lowercase letter `o`. To open up a new line above the current line, hit the uppercase letter `O` (`SHIFT+O`). Note that, in both cases, you'll be placed in Insert Mode.

`o`	**o**pen	Open new line below current line and enter Insert Mode.
`O`	**O**pen	Open new line above current line and enter Insert Mode.

To copy-and paste the current line (where the cursor is located), hit the lowercase letter `y` twice: `yy`. This will silently make a copy of the line and place it in the `vi` Editor buffer. The letter `y` stands for yank. Hitting the lowercase letter `p` will paste the yanked line in the buffer into the editor window. Note that you can repeatedly hit the letter `p`. Do that now to create `10` duplicated lines in your file.

`yy`	**y**ank	Yank (copy) the current line to the buffer.
`p`	**p**aste	Paste the line from the buffer into the editor window (below).
`P`	**P**aste	Paste the line from the buffer into the editor window (above).

Now that you have `10` lines in your file, you can either use the up-arrow to go up to the first line, or the down-arrow to navigate to the last line. But, if you have many lines, this will take some time…and *ain't nobody got time fo' dat*.

To navigate to the first line, hit the number `1` followed by the uppercase letter `G` (`1,SHIFT+G`). The number `1` indicates Line #`1`, but you can substitute any number prior to hitting `SHIFT+G`. To go to Line #`9`, just hit `9` then `SHIFT+G`. To go to the bottom of the file, just hit the uppercase letter `G` (`SHIFT+G`).

`1g`	**g**o	Move the cursor to Line #`1`.
`G`	**G**orge?	Move the cursor to the gorge (last line) of the file.

To move the line just below the cursor's location onto the current line, hit the uppercase letter `J` (`SHIFT+J`). Note that you can continually hit `SHIFT+J` to join subsequent lines to the current line.

`J`	**J**oin	Join the next line to the current line.

You can overwrite a single letter by hitting the lowercase letter `r` followed by your replacement character. You can do continuous overwriting by hitting the uppercase letter `R` (`SHIFT+R`). Note that `SHIFT+R` will display the indication `-- REPLACE --` at the bottom of the screen (where you normally see `-- INSERT --`). Hit the ESC key to exit to Command Mode.

`r`	**r**eplace	Replace a single letter.
`R`	**R**eally replace	Continuous replacement. Hit ESC to exit into Command Mode.

To save your changes, hit `SHIFT+:`, type the lowercase letter `w` and hit the Enter key. At the bottom of the editor you should see an indication that your file was saved (`10` lines (`L`) containing `510` characters (`C`) written to the file).

```
~
"bobsmith.txt" [New] 10L, 510C written
```

To save and quit the `vi` Editor at the same time, hit `SHIFT+:`, type the lowercase letters `wq` and hit the Enter key. Alternatively, you can hit the uppercase letter `Z` twice (`SHIFT+ZZ`) to save and quit. To quit the `vi` Editor **without saving any changes**, hit `SHIFT+:`, type the lowercase letter `q!` and hit the Enter key. If you forget the exclamation point, but you did make some changes to the file, the `vi` Editor will display a warning indicating that you can override by putting the exclamation point in.

`SHIFT+:+w`	**w**rite	Save the file.
`SHIFT+:+wq`	**w**rite and **q**uit	Save the file then exit the `vi` Editor.
`SHIFT+ZZ`	**w**rite and **q**uit	Save the file then exit the `vi` Editor (*coolest method ever!*).
`SHIFT+:+q!`	**q**uit w/o save	Exit the `vi` Editor without saving the file.
`SHIFT+:+q`	**q**uit	Exit the `vi` Editor, but a warning message will be displayed (if changes made).

You can insert an external file into the file you're currently editing by hitting `SHIFT+:+r` followed by the name of the external file. At the bottom of the editor, you'll see something like this:

```
~
~
~
:r bobsmith2.txt
```

Don't confuse this `:r` command with the single character replace command `r` when in Command Mode. Now, if you feel you want to save the current file to a different file, as is your right and privilege as a human being, you can use `SHIFT+:+w`, but followed up by the name of the new file. If the file exists and you want to overwrite it, replace `w` with `w!`. At the bottom of the editor, you'll see something like this:

```
~
~
~
:w bobsmith2.txt
```

Note that you're **not** automatically moved to the new file, but remain in the current file!

`SHIFT+:+r` *`file`*	**r**ead *`file`* into	Read external *`file`* into current file.
`SHIFT+:+w` *`file`*	**w**rite to *`file`*	Save current file to *`file`*.
`SHIFT+:+w!` *`file`*	**w**rite to *`file`*	Save current file to *`file`* with overwrite.

Note that you can combine several of the Command Mode actions above to perform extended actions. For example, recall that you can hit `yy` and the current line will be placed in the buffer. But, you can yank the next *`n`* line by hitting *`n`*`yy`. For example, `3yy` yanks the current line as well as the following two lines (three total lines) and

places them in the buffer. The vi Editor will give you an indication of the number of lines yanked at the bottom of the screen:

```
~
~
~
3 lines yanked
```

You can yank all of the lines from the current line to the end of the file by hitting the lowercase letter y followed by the uppercase letter G (y+SHIFT+G). Again, the number of lines yanked will be indicated at the bottom of the editor.

You can yank a single word by hitting the lowercase letters y and w: yw. Of course, you can yank several words by hitting *n*yw by specifying the number of words. Three words: 3yw.

*n*yy	**y**anks *n* lines	Yank *n* lines to the buffer.
yG	**y**ank to **G**orge	Yank from current line to the end of the file.
yw	**y**ank **w**ord	Yank a word to the buffer.
*n*yw	**y**anks *n* words	Yank *n* words to the buffer.

Now, the combinations shown above are fairly limited, so the vi Editor has the ability to **mark** lines in the file. When in Command Mode (punch that ESC button, pal!), hit the lowercase letter m followed by a mark name. I generally use the letters a and b, but I'm boring. For example, move to Line #1 in the file, and hit ma. Then move to Line #5 in the file, and hit mb. Note that the vi Editor will silently set these marks and no indication will appear on the screen. You can now move between the two marks by hitting the apostrophe followed by the mark name: 'a or 'b. Doing this will move your cursor between the two marks. We can now combine the apostrophe, the mark names and the yank action y to yank all of the lines between mark a and mark b, inclusive, to the buffer: 'ay'b. You should see something like this at the bottom of the screen:

```
~
~
~
10 lines yanked
```

And, you guessed it, if you want to paste these yanked lines somewhere else, just move your cursor to the line above where you want to paste them and hit the lowercase letter p. You can use the uppercase letter P (SHIFT+P) to paste above the current line.

ma	**m**ark a	Marks the current line as a.
'a	move to mark a	Move cursor to the line marked as a.
'ay'b	**y**ank **from** a **to** b	Copy all lines between marks a and b, inclusive.

Occasionally, you'll want to shift some code over one or more columns to the right or left. First, set the marks a and b to the beginning and end of the code block you want to shift. Combining the apostrophe, the mark names and the shift right action symbol **>**, you can move your code over to the right: 'a>'b. Of course, you can move over to the left using the shift left action symbol 'a<'b. By default, text will shift by one tab. I normally reset this to one space. To do this, click SHIFT+: followed by the command set shiftwidth=1, and hit the Enter key:

```
~
~
~
:set shiftwidth=1
```

Note that you can substitute the letters sw for shiftwidth in the set above.

Now, when you perform `'a>'b` or `'a<'b`, your code will move one column (not tab) at a time, either right or left. Note that rather than repeating the actions `'a>'b` or `'a<'b` multiple times to move code over more than one column, you can repeat the previous action by hitting period (`.`) several times instead. Nice!

`'a>'b`	shift right	Shift lines from `'a` to `'b` to the right.
`'a<'b`	shift left	Shift lines from `'a` to `'b` to the right.
`:set shiftwidth=1`	change shift width	Reset `shiftwidth` to `1` column.
`.`	repeat	Repeats previous command.

On a more mundane note, if you'd like to search through the file, hit the slash (`/`) key, followed by the search text, followed by the Enter key. You can move to the next occurrence by hitting the lowercase letter `n`, and to the previous occurrence by hitting the uppercase letter `N` (`SHIFT+N`).

`/text`	search for `text`	Search for `text` in the file.
`n`	**n**ext	Search for the next occurrence of `text`.
`N`	pre**N**ious?	Search for the previous occurrence of `text`.

If you're into data – and who isn't! – you can hit `CTRL+G` to display the name of the current file, whether the file has been modified or not, the total number of lines in the file, and where your cursor is located as a percent of total line, like this:

```
~
~
"bobsmith.txt" [Modified] 10 lines --20%--
```

Note that hitting `SHIFT+G` just moves your cursor to the bottom of the file.

`CTRL+G`	file info	Display file info.
`SHIFT+G`	move to **G**orge	Move cursor to the end of the file.

If you'd like to alter the text either globally or between two marks, you can use the `SHIFT+:` command line with actions similar to the stream editor (`sed`) we talked about earlier. Let's change the word `antelope` to `dinosaur` across the entire file. Hit `SHIFT+:`, followed by this text:

```
1,$ s/antelope/dinosaur/g
```

Here, the `1` indicates the first line and the dollar sign (`$`) indicates the last line. Taken together `1,$` indicates the entire file. Again, similar to sed, `s/`*`old`*`/`*`new`*`/g` indicates you want to replace the `old` text with the `new` text globally (`g`). When you hit the Enter key, all occurrences of `antelope` are replaced with `dinosaur`, and rightly so as antelope are vicious, mean-spirited ruffians. Note that the editor will indicate the number of changes at the bottom of the screen:

```
~
~
~
10 substitutions on 10 lines
```

You can undo these changes by clicking the lowercase letter `u`. You can continue to hit the letter `u` to undo previous changes, but continuing to hit the letter `u` will eventually delete the entire universe, so caution is advised.

Now, you can replace the line range `1,$` with two set marks (say, `a` and `b`), like this:

```
'a,'b s/antelope/dinosaur/g
```

`1,$ s/`*`text-1`*`/`*`text-2`*`/g`	global replace	Replace *`text-1`* for *`text-2`* across entire file.
`'a,'b s/`*`text-1`*`/`*`text-2`*`/g`	limited replace	Replace *`text-1`* for *`text-2`* between two marks.

Finally, to flip the case of a letter, hit the tilde key (~). If you continue to hit the tilde key, your cursor will move automatically to the next letter and change its case.

~	change case	Changes the case of one or more characters.

Updating the `.bash_profile` File

Now that we have the `vi` Editor under our belt, let's modify the file `.bash_profile` in the `/home/smithbob` directory. Note that this file is executed every time you log into your account and is a great place to put the two aliases: `lsf` and the hot-wired `rm` command. So, go back to your home directory, edit the file `.bash_profile` using the `vi` Editor, move to the very bottom of the file (`SHIFT+G`) and add the following two lines to the end of it (hit `o` to open a new line and enter Insert Mode):

```
alias lsf="ls -alF"
alias rm="rm -i"
```

Next, save and quit out of the file (hit ESC to get back into Command Mode, hit `SHIFT+:` followed by the letters `wq` and hit the Enter key to save the file and quit out of the `vi` Editor). Now, just modifying the file won't alter your current session. To update your current session with the changes you just made in `.bash_profile`, use the `source` command followed by the name of the file:

```
[smithbob@lnxserver ~]$ source .bash_profile
```

You can now try the commands `lsf` and `rm` at your leisure. You can also see all of the aliases available in your current session by entering the command `alias` and hitting the Enter key:

```
[smithbob@lnxserver ~]$ alias
alias lsf='ls -alF'
alias rm='rm -i'
[smithbob@lnxserver ~]$
```

Updating the `.vimrc` File

Note that the `vi` Editor allows you to place `set` commands in a file called `.vimrc` in your home directory. The entries in this file will be executed each time you start the `vi` Editor. This prevents you from having to, say, set the `shiftwidth` each time you want it to be `1`. Let's test this out! Go to your home directory and open the file `.vimrc` using the `vi` Editor. Note that this file may not exist, so you'll just be creating a new file. Next, type in the following line:

```
set shiftwidth=1
```

Save and exit out of this file.

Next, start the `vi` Editor again, in Command Mode type `:set` and hit the Enter key. You should see something similar to the following. Note that `set` just shows your current settings for things like, say, `shiftwidth`.

```
:set
--- Options ---
  cscopetag          helplang=en        scroll=18          ttyfast            window=36
  cscopeverbose      hlsearch           shiftwidth=1       ttymouse=sgr       t_Sb=^[[4%dm
  filetype=text      ruler              syntax=text        viminfo='20,"50    t_Sf=^[[3%dm
  backspace=indent,eol,start
  cscopeprg=/usr/bin/cscope
  fileencodings=ucs-bom,utf-8,latin1
  guicursor=n-v-c:block,o:hor50,i-ci:hor15,r-cr:hor30,sm:block,a:blinkon0
Press ENTER or type command to continue
```

As you see above, `shiftwidth` is set to the value of `1` now thanks to the entry we placed in the `.vimrc` file.

Occasionally, when you open a file containing long lines of text, the text will be wrapped around in the `vi` Editor. Now, the `vi` Editor doesn't *physically* wrap the lines in the file, it's just how these lines are *displayed*. If you don't want the `vi` Editor to do that, you can set the option `nowrap`. While in Command Mode, type `:set nowrap` and hit the Enter key. You'll immediately notice that the long lines are no longer wrapped. Naturally, you can add this to your `.vimrc` file so that it takes affect each time you start the `vi` Editor. Once the text is no longer wrapped, you can *visually* slide the text to the **L**eft by hitting `zL` one or more times. To slide back to **H**ome, you hit `zH` one or more times. To undo any sliding, just hit the (beginning of line) `0` key.

`:shiftwidth`	moves # chars	Moves lines left of right a specified number of columns.
`:nowrap`	no line wrapping	Prevents long lines from being wrapped in the editor.
`zL`	slide data **L**eft	Slides the displayed data **L**eft. Can repeat the command as needed.
`zH`	slide display **H**ome	Slides the displayed text back **H**ome. Can repeat the command as needed.

Chapter 20 – Working with Linux Scripts

While you may think that running Linux commands from the command prompt is the bee's knees, we can do much better by placing our commands in a file called a *script* and then running, or executing, it from the command prompt. Linux scripts contain a series of commands just like those we discussed earlier, but can also contain commands used to, say, run a Python program, or run a fancy-shmancy statistical analysis with R, or query the database using ImpalaSQL using the command line utility `impala-shell`, or query the database using HiveQL using the command line utility `beeline`, and so on. Really, the possibilities are endless…(endless: just like my run-on sentences).

Now, a script isn't just a neat and tidy place to put your commands, but can be scheduled to run at a certain day and/or time, either once or repeatedly. Say, for example, every Sunday night at 11:00 PM you want a particular Hadoop table to be updated with the latest data from your legacy database, or info from a specific web site, or with data pulled down via an API, etc. You create your script, test it, and then schedule it to run every Sunday night at 11:00 PM, say. Done and dusted! We discussed scheduling jobs in *Chapter 31 – Scheduling Jobs Using crontab*.

Recall, in the previous chapter, we talked about the Linux `chmod` command. We make use of this command to make scripts *executable*. If this step is skipped, Linux won't know what you're talking about, your script won't run, and war will break out in Bratislava. To mark a script as executable, first create a file called `myscript1`, say, by either using `touch` or with the `vi` Editor, then run the following command at the command line:

```
[smithbob@lnxserver ~]$ chmod u+x myscript1
```

This command will make the script `myscript1` e**x**ecutable for the **u**ser only; that is, you'll be able to run it from the command line and the suckers you work with won't. MWA HA HA!

A Bouncing Baby Script

Let's create a simple script, called `myscript1`, which places the contents of `/proc/cpuinfo` into a log file. Along the way, we'll use the `echo`, `cat`, and `date` commands just to make everything pretty. Using the `vi` Editor, edit the file `myscript1` and type in the following text (hit `i` to enter Insert Mode, dude!):

```
#!/bin/bash

# Create our log file by using echo with some text and the date command.
echo "Program started at `date`." > /home/smithbob/myscript1.log

# Append the contents of /proc/cpuinfo to the log file.
cat /proc/cpuinfo >> /home/smithbob/myscript1.log

# Close out our log file by appending some final text.
echo "Program ended at `date`." >> /home/smithbob/myscript1.log

exit
```

Once you've finished entering the text, save and exit out of the `vi` Editor (ESC to enter Command Mode, `SHIFT+:wq` and smoosh the Enter key with a hearty gusto never before seen in a low-budget book such as this).

Here's the low-down on this code:

1. The top line indicates which program will be used to execute the subsequent text. While you may be used to running programs by typing the name of the executable at the command line (e.g., `python`, `sas`, etc.) followed by the name of the file containing code to be run in that language, for Linux scripts the program that's run is called `bash` and is located in the `/bin` directory. As a side note, you can replace the text `/bin/bash` with the directory/name of any other executable. For Python, say, you can replace Line #1 above with the following…

```
#!/usr/bin/python
```

…to execute the Python code stored within the same file. I don't normally do this because I prefer to keep my code separate from the script...call me a rebel.

2. The first `echo` command creates a log file called `myscript1.log` containing the text **`Program started at`** followed by the current date and time. Here, we're making use of the backticks (` `` `) to run the `date` command behind the scenes. Also note that we're using a single greater than symbol (>) to ensure that any older version of the log file is overwritten.
3. The `cat` command dumps the contents of `/proc/cpuinfo` into the log file. Take note that we're using two greater than symbols (>>) to append to our log file. Both > and >> are the redirection arrows we talked about earlier.
4. The second `echo` command appends the text **`Program ended at`** followed by the `date` command again in backticks (` `` `).
5. Finally, the `exit` command just ends the script. Take note that the `exit` command returns the return code of the last command executed in the script, in this case, the `echo` command. We talk more about how to capture this return code later in the book.

As indicated above, we need to make our script executable. At the command line, issue the following code:

```
[smithbob@lnxserver ~]$ chmod u+x myscript1
```

Now, in order to execute the script from the command line, type in a period, followed by a forward slash, followed by the name of the script:

```
[smithbob@lnxserver ~]$ ./myscript1
```

Note that the **`./`** just indicates that the script is to be run from the current directory (or, as I like to call it: *here*).

At this point, your script will run nearly instantaneously and you will, once again, be staring at the desolate sand dune that is the Linux command prompt:

```
[smithbob@lnxserver ~]$ sand dune
```

Now, if you list the contents of the directory (`lsf`), you'll see the log file `myscript1.log`. Using the `head` and `tail` commands, let's see the top and bottom five lines in the log file:

```
[smithbob@lnxserver ~]$ head myscript1.log
Program started at Mon Nov  8 14:04:42 EST 2021.
processor   : 0
vendor_id   : GenuineIntel
cpu family  : 6
model       : 78
model name  : Intel(R) Core(TM) i7-6500U CPU @ 2.50GHz
stepping    : 3
cpu MHz           : 2591.786
cache size  : 4096 KB
physical id : 0

[smithbob@lnxserver ~]$ tail myscript1.log
wp          : yes
flags       : fpu vme de cx8 ...snip... invpcid rdseed clflushopt
bugs        :
bogomips    : 5183.57
clflush size      : 64
cache_alignment   : 64
address sizes     : 39 bits physical, 48 bits virtual
power management:
```

```
Program ended at Mon Nov  8 14:04:42 EST 2021.
```

As you see, the notes we placed in the log file using the `echo` command appear along with the contents of `/proc/cpuinfo`.

Hulk Smash! Hulk Crash! Hulk `bash`!

As we indicated in the previous section, Linux scripts are run by a program called `bash`, located in the `/bin` directory. In fact – and you probably want to be sitting down for this – everything at the Linux command line is run using `bash`. Yep! Every command you type in is run by `bash`. `bash` is generically known as a *shell* and is what you see when you log in using PuTTY. It's responsible for running your commands as well as Linux scripts. But, it's not the only shell available, although it's the only one you need be concerned with for now. If you ever work on another Unix or Linux operating system, you may well be using a completely different shell from `bash`, like `ksh` or `csh`. I mention this because if you ever need to look up syntax online, you'll know to stick to `bash`. Also, there are several books written, and songs sung, about `bash`. *Tra-la-la!*

The Scripts Are Takin' Over!

Because the script `myscript1` runs so quickly, you may not have noticed that the command prompt is *blocked* momentarily, preventing you from entering in any additional commands; that is, your script literally takes over your command line session. There are several methods you can employ to do the tango around this:

1. You can run your script *in the background* freeing up the command prompt to respond to more lovely commands. You do this by placing an ampersand (`&`) after the name of the script:

   ```
   ./myscript1 &
   ```

2. If you run your script in the background using Method #1 above, and then quickly log out, your script will automatically be killed. You can prevent this most unnecessary murder by adding the command `nohup` before the name of your script as well as placing an ampersand (`&`) after it:

   ```
   nohup ./myscript1 &
   ```

   The command `nohup` stands for *no hang up* and will prevent your job from being killed when you log out. No dead corpses here!

For example, using Method #1, let's run our script in the background:

```
[smithbob@lnxserver ~]$ ./myscript1 &
[1] 7120
[smithbob@lnxserver ~]$
[1]+  Done                    ./myscript1
```

Take note that when you run a script in the background, as shown above, a number will be displayed and, in our case, is the number `7120`. No, silly, Linux isn't recommending a lottery number, it's the *process ID number* (PID) and is very useful if you have to kill your running job (see below). Note that if you hit the Enter key a few times, you'll be told that the script has completed as indicated by the word `Done` along with the name of the script. Naturally, depending on how long your script runs, it may take a while before you see the word `Done`.

Now, let's try Method #2:

```
[smithbob@lnxserver ~]$ nohup ./myscript1 &
[1] 7129
[smithbob@lnxserver ~]$ nohup: ignoring input and appending output to
                                                                `nohup.out'
[1]+  Done                    nohup ./myscript1
```

Very similar to Method #1, but Method #2 tells you that any output will be appended to the file called `nohup.out`. This file, usually located in the same directory to where you executed the script, contains output produced by any command executed from your script that you didn't capture using redirection, say, to a log file.

Although we discussed many Linux commands in the previous chapter, there are a few more we have to talk about, specifically, having to do with finding a running script and then unceremoniously killing it. When you submit a script using either Method #1 or Method #2, you're given a process ID number (PID). You can determine all of your currently running jobs by running the command `ps` (process status) from the command line:

```
[smithbob@lnxserver ~]$ ps
  PID TTY          TIME CMD
 5526 pts/0    00:00:00 bash
 7781 pts/0    00:00:00 ps
[smithbob@lnxserver ~]$
```

`ps`	**p**rocess **s**tatus	Displays all running processes associated with **your** current session.
`ps -ef`	**p**rocess **s**tatus	Display **all** running processes on the server.

In the output above, you're shown all currently running processes for **your** Linux session. Since the script `myscript1` finished ages ago and is now microwaving salmon in the company breakroom, we're only presented the two entries shown above. The two most important columns are `PID` and `CMD`. The `PID` column indicates the process ID number and is useful if you need to kill a job. The `CMD` column displays the abbreviated names of running scripts and is useful if you've submitted several jobs at once. But, be aware that `ps` displays **your own processes** and nothing more. If you want to see the full list of running processes on the Linux server, run the command `ps -ef`. Piping the output from this command into `grep` allows you to search for running scripts by user name, executable name, etc.:

```
[smithbob@lnxserver ~]$ ps -ef | grep "smithbob"
smithbob    5526  5522  0 13:19 pts/0    00:00:00 bash
smithbob    7973  5526  0 15:06 pts/0    00:00:00 grep bash
[smithbob@lnxserver ~]$
```

In the output above, the `PID` column is the second column (`5526` and `7973`, in this case). Since we're using the `grep` command, the lovely header row is not displayed since the text `smithbob` does not appear on the header row. (Bob Smith should really file a complaint!)

Now, if you ever want to kill a job, you can use the `kill` command followed by the `PID`.

`kill` *pid*	**kill**	Kills a specific process using the process ID number (PID).

For example, let's assume the script `myscript1` is taking way too long hacking into a foreign leader's Swiss bank account and you want to kill it. If you submitted the script in the current session, you can use `ps` to find the `PID`. If you submitted the script, logged out, went home, watched an episode of *Squid Game*, went to sleep, couldn't sleep because *Squid Game* is scary, finally got to sleep, woke up, took a shower, drove into work and logged in, you can use `ps -ef` along with the `grep` command to find the `PID` (assuming the job is still running). In any case, let's kill the running job with a process ID of `1234`:

```
[smithbob@lnxserver ~]$ kill 1234
```

This is usually enough to murderize the script, but if you run the appropriate variation of `ps` and notice the script isn't completely stone dead, but more walking dead, you may need to be more brutal. In these instances, you can add the switch `-9` to the kill command:

```
[smithbob@lnxserver ~]$ kill -9 1234
```

This will kill the job dead, *Player 456*!

Don't Forget to Redirect STDERR

Recall I mentioned that you can redirect output from the standard error stream (`STDERR` aka `2`) so you don't lose informative error and warning messages. Let's modify the script `myscript1` to ensure all error messages produced by the `cat` command are placed in the log file:

```
cat /proc/cpuinfo >> /home/smithbob/myscript1.log 2>&1
```

Now, it's unlikely the simple `cat` command will produce error messages, so the example above is very silly. But, errors may appear while running SQL scripts using the commands `impala-shell` or `beeline`, or while running a Python program using the command `python`, etc., and these error messages should be captured in the log file.

`tl;dr`: Redirect the `STDERR`.

Simultaneously Parallel Processing Concurrently at the Same Time

At first blush, it looks like a standard Linux script just processes commands one at a time waiting for each to finish before moving on to the next. This is true, but you can set up your script to run several commands concurrently and then wait for all of them to finish before moving on. Recall that programming languages such as Python and R are not set up to perform parallel processing *outta da box* (although there are several packages which can run code in parallel). One way around this is to just execute several Python, R, ImpalaSQL, Java, etc. programs concurrently; that is, at the same time.

In your Linux script, place an ampersand (&) at the end of each line you want to run in parallel. This is akin to Method #1 we talked about earlier. Note that `nohup` is not needed here. For example, let's submit four ImpalaSQL programs to run concurrently:

```
impala-shell -i hdpserver --database prod_schema -f /home/smith/query1.sql &
impala-shell -i hdpserver --database prod_schema -f /home/smith/query2.sql &
impala-shell -i hdpserver --database prod_schema -f /home/smith/query3.sql &
impala-shell -i hdpserver --database prod_schema -f /home/smith/query4.sql &
wait
```

Take note that the last line is the `wait` command. This will force the script to wait (i.e., *block*) until all four of the submitted jobs have completed. We talk more about the `impala-shell` command later in the book.

And, as stated in the previous section, don't forget to redirect any output as well as the `STDERR`. I didn't include that in the code above due to space constraints (unless you want me to use 6-point font, bruh!).

But, be careful not to submit soooooooo many parallel programs that you bring down the edge node server. Take into account the number of processors available on your edge node server as well as the normal activity on that server. Start off with a few parallel programs and monitor the CPU activity using the Linux command `top`, which is akin to the Windows Task Manager's Processes tab. Note that `top` does take over your PuTTY screen, but you can easily quit out by hitting the letter `q`. Here's an example of the output `top` displays:

```
top - 14:56:27 up 3 min,  2 users,  load average: 2.13, 1.59, 0.68
Tasks: 270 total,   1 running, 269 sleeping,   0 stopped,   0 zombie
Cpu(s): 10.9%us,  3.2%sy,  0.0%ni, 85.6%id,  0.2%wa,  0.0%hi,  0.2%si,  0.0%st
Mem:   7911572k total,  3894092k used,  4017480k free,    50620k buffers
Swap:  5324796k total,        0k used,  5324796k free,   551488k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 5161 oracle    20   0  472m  24m  20m S  6.3  0.3   0:00.56 gnome-panel
 3372 yarn      20   0 2699m 315m  28m S  3.6  4.1   0:14.18 java
 4500 root      20   0  266m  50m  21m S  3.3  0.6   0:03.52 Xorg
 5444 oracle    20   0  290m  22m  19m S  2.6  0.3   0:01.04 gnome-terminal
 2813 hdfs      20   0 2782m 286m  29m S  2.0  3.7   0:12.38 java
 4084 spark     20   0 3532m 307m  42m S  2.0  4.0   0:09.31 java
```

```
 2695 hdfs      20   0 2750m 234m  28m S  1.3  3.0   0:10.84 java
 3149 mapred    20   0 2773m 262m  29m S  1.3  3.4   0:11.97 java
 3250 yarn      20   0 2573m 327m  29m S  1.3  4.2   0:11.95 java
 4328 impala    20   0 5485m 167m  54m S  1.3  2.2   0:09.30 impalad
 3880 hive      20   0 3930m 382m  51m S  1.0  4.9   0:16.40 java
 4204 impala    20   0  359m  28m  24m S  0.7  0.4   0:00.50 statestored
 4249 impala    20   0 3729m 164m  47m S  0.7  2.1   0:09.02 catalogd
 5370 oracle    20   0  296m  18m  16m S  0.7  0.2   0:00.14 multiload-apple
 5463 oracle    20   0 15224 2140 1704 R  0.7  0.0   0:00.18 top
    7 root      20   0     0    0    0 S  0.3  0.0   0:00.42 rcu_sched
 1869 dbus      20   0 30556 2940 2000 S  0.3  0.0   0:00.42 dbus-daemon
 2066 root      20   0 22576 1036  912 S  0.3  0.0   0:00.26 hald-addon-inpu
```

The `COMMAND` column indicates the running command and you should be able to determine your programs from it. Take note of the `%CPU` and `%MEM` during the execution of your script and ensure your programs are not dragging down the server.

While in `top`, if you hit the number `1` key, you'll see a full list of the processors available on the edge node server, shown below:

```
top - 14:58:12 up 5 min,  2 users,  load average: 0.45, 1.14, 0.62
Tasks: 256 total,   2 running, 254 sleeping,   0 stopped,   0 zombie
Cpu0  : 10.4%us,  2.2%sy,  0.0%ni, 87.4%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  5.7%us,  1.8%sy,  0.0%ni, 92.2%id,  0.4%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   7911572k total,  3895952k used,  4015620k free,    50924k buffers
Swap:  5324796k total,        0k used,  5324796k free,   551644k cached
```

Hit the letter `q` to quit `top`.

The column of percentages appearing first, `10.4%us` and `5.7%us`, indicate the current usage on each CPU.

`top`	**t**able **o**f **p**rocesses	Displays CPU, RAM, etc. information akin to Windows Task Manager.

Oh, and you may want to run your code at night when your colleagues are in their jimmy-jams sleepy-bobo.

Using Variables in Scripts

You can create and use variables within your own scripts. Similar to most other programming languages, you create a variable by specifying the variable's name followed by an equal sign. If you want to set the variable to some text, surround that text with double-quotes. If, instead, you want to set the variable to a number, forget the double-quotes, just go numeric commando. For example, let's create the following variable containing the URL of Taco Bell's website:

```
sTacoBell="http://www.tacobell.com"
```

And, let's create a variable to hold the incredulously large total number of Taco Bell locations worldwide:

```
sTacoBellLocCnt=7072
```

Now, if you want to use these variables in your code, place a dollar sign before the name of the variable:

```
echo "The Taco Bell website is $sTacoBell and there are $sTacoBellLocCnt
locations worldwide."
```

You can also resolve variables using the following beautiful curly brace syntax:

```
echo "The Taco Bell website is ${sTacoBell} and there are ${sTacoBellLocCnt}
locations worldwide."
```

Now, don't forget the use of the backticks (``) to run a Linux command in the background within your script. For example, let's create a variable to hold the current four-digit year:

```
sYYYY=`date +%Y`
echo $sYYYY
2021
```

The use of the backticks with variables can be very helpful especially if you want to pull a specific value from the Hadoop database using, say, the `impala-shell` command line utility. For example, let's create a variable in our script which contains the number of rows in the `prod_schema.dim_us_state_code` table from a query executed directly on the database (see the `-q` switch below):

```
sRowCnt=`impala-shell -B --database=default
                      -q 'select count(*) from
                                              prod_schema.dim_us_state_mapping;'`
echo $sRowCnt
65
```

Take note that I'm using the `-B` switch with `impala-shell`. This removes the header and formatting box around the output, which isn't needed in this case since we just want the number `65`. *Boxes! We don't need no stinkin' boxes!* We talk more about the `impala-shell` command in *Chapter 21 – Running ImpalaSQL from the Linux Command Line*.

Using `if-then-else`

As with most programming languages, Linux scripts allow for the `if-then-else` construct and varieties thereof. For example, the general `if-then` statement looks like this:

```
if [ condition ]
then

 statements

fi        WHOA!! END OF if!
```

And, the `if-then-else` statement looks like this:

```
if [ condition ]
then

 statements

else

 statements

fi        WHOA!! END OF if!
```

And, finally, you can have multiple `else`/`if` statements. Note that `elif` indicates an impeding `else if` condition.

```
if [ condition ]
then

 statements

elif [ condition ]
```

```
 statements

elif [condition ]

 statements
...

else

 statements

fi <------- WHOA!! END OF if!
```

Similar to other `if`-`then`-`else` constructs, you can test a variety of conditions within the square brackets:

`sA = sB`	Tests if the string `sA` is equal to the string `sB`.
`sA == sB`	Tests if the string `sA` is equal to the string `sB` .
`sA != sB`	Tests if the string `sA` is not equal to the string `sB` .
`I -eq J`	Tests if the integer `I` is equal to the integer `J`.
`I -ne J`	Tests if the integer `I` is not equal to the integer `J` .
`I -gt J`	Tests if the integer `I` is greater than the integer `J` .
`I -lt J`	Tests if the integer `I` is less than the integer `J` .
`I -ge J`	Tests if the integer `I` is greater than or equal to the integer `J` .
`I -le J`	Tests if the integer `I` is less than or equal to the integer `J` .

Take note that `condition` is surrounded by square brackets (`[ condition ]`), but on some Linux flavors you may need to double that up (**`[[`** *`condition`* **`]]`**)…because more is better!

As an example, let's test a variable for equality:

```
iNUM=2
if [ $iNUM -eq 2 ]
then
 echo "MATCHED IT."
else
 echo "DIDN'T MATCH IT."
fi
MATCHED IT.
```

Using the `case` Statement

The simple `case` Statement is available in Linux scripts and has the following syntax:

```
case $var in
 match1)
  statement-1₁
  statement-1₂
  ...
 ;;
 match2)
  statement-2₁
  statement-2₂
  ...
 ;;
 *)
  statement-n₁
  statement-n₂
  ...
```

```
 ;;
esac <———— WHOA!! END OF case!
```

Note that the final match, indicated by `*)`, is similar to an `else` condition and is executed if `$var` doesn't match anything. For example,

```
iNUM=2
case $iNUM in
 2)
  echo "MATCHED IT."
 ;;
 *)
  echo "DIDN'T MATCH IT."
 ;;
esac <———— WHOA!! END OF case!
MATCHED IT.
```

When using text, surround the matches by double-quotes.

Using Logical Operators

You can use logical operators in your `if-then-else` statements:

&&	Logical And	And's two conditions.
||	Logical Or	Or's two conditions.
!	Negation	Negates a condition.

```
iNUM=2
sSTATUS="GO"
if [[ $iNUM -eq 2 && "$sSTATUS"=="GO" ]]
then
 echo "MATCHED AND WE ARE A-GO."
else
 echo "LAUNCHED DELAYED."
fi
MATCHED AND WE ARE A-GO.
```

Using Loops

Similar to other programming languages, Linux scripts can perform loops with the `for`, `while` and `until` statements. For example, let's use the `for` statement to perform a loop `10` times:

```
for iINDX in {1..10}
do
 echo "Index number is currently $iINDX."
done <———— WHOA!! It's not backwards!! Go figure!

Index number is currently 1.
Index number is currently 2.
Index number is currently 3.
Index number is currently 4.
Index number is currently 5.
Index number is currently 6.
Index number is currently 7.
Index number is currently 8.
Index number is currently 9.
Index number is currently 10.
```

Take note that the starting value is 1 and the ending value is 10 with two periods (..) between them. If you would like to skip values, you specify an additional two periods (..) followed by the skip value. For example, let's re-do the example above, but skipping by two:

```
for iINDX in {1..10..2}
do
 echo "Index number is currently $iINDX."
done

Index number is currently 1.
Index number is currently 3.
Index number is currently 5.
Index number is currently 7.
Index number is currently 9.
```

Note that the starting value can be greater than the ending value. In this case, the loop will count down:

```
for iINDX in {10..1..2}
do
 echo "Index number is currently $iINDX."
done

Index number is currently 10.
Index number is currently 8.
Index number is currently 6.
Index number is currently 4.
Index number is currently 2.
```

Next, let's re-create the first `for` loop using a `while` statement:

```
iINDX=1
while [ $iINDX -le 10 ]
do
 echo "Index number is currently $iINDX."
 ((iINDX++))
done
```

It's worth noting that there are several approaches to incrementing or decrementing the index variable `iINDX` in the code above. As written, you can easily add 1 to `iINDX` by coding the double-plus signs (++) after the variable name. To decrement by 1, use double-minus signs (--) instead. Alternatively, you can increment or decrement by a specific value by using the += or -= constructs:

```
iINDX=1
while [ $iINDX -le 10 ]
do
 echo "Index number is currently $iINDX."
 ((iINDX+=1))
done
```

Of course, you can go full-on old school:

```
iINDX=1
while [ $iINDX -le 10 ]
do
 echo "Index number is currently $iINDX."
 ((iINDX=iINDX+1))
done
```

In all cases, you must surround the increment/decrement code with two sets of parentheses.

For a lovely change of pace, you can use the `until` statement instead of the `while` statement:

```
iINDX=1
until [ $iINDX -gt 10 ]
do
 echo "Index number is currently $iINDX."
 ((iINDX=iINDX+1))
done
```

While performing your loops, you may want to exit the loop if some magical condition is met. In this case, you can use the `break` statement to break out of the `while` loop:

```
iINDX=1
while [ $iINDX -le 10 ]
do
 echo "Index number is currently $iINDX."
 ((iINDX++))

 if [ $iINDX -eq 7 ]
 then
  echo "LUCKY NUMBER SEVEN"
  break
 fi

done
```

Similar to the `break` statement, you can force a loop to immediately move on to the next iteration by specifying the `continue` statement.

```
for iINDX in {1..10}
do

 if [ $iINDX -eq 7 ]
 then
  continue
 fi

 echo "Index number is currently $iINDX."

done
```

Passing Parameters into a Script

You can pass parameters into your Linux scripts from the Linux command line. Let's create a Linux script called `querydb` which accepts three parameters: the name of a SQL query file, the starting year and the ending year. Here's a simplified version:

```
#!/bin/bash

sALL_ARGS="$@"
sSQLCode="$1"
iBEGYYYY="$2"
iENDYYYY="$3"

echo "Querying database with SQL code $sSQLCode starting from year $iBEGYYYY
and ending at year $iENDYYYY."

exit
```

Note that the first parameter on the command line is automatically named `$1`, the second `$2`, and so on. If your parameter contains spaces, then place it entirely within quotes so the script will consider it as one parameter, not multiple parameters. All of the parameters are placed in one variable named **`$@`**. Also, the name of the script is placed in the variable `$0`. Here's an example call to the program:

```
[smithbob@lnxserver ~]$ ./querydb update.sql 2020 2021
```

When the script executes, the following is output:

```
Querying database with SQL code update.sql starting from year 2020 and ending
at year 2021.
```

And, I realize I'm harping on the backticks (``) thingy, but you can use them at the command line as parameters to your scripts as well (as usual, everything should appear on a single line, but shown here on separate lines for clarity):

```
[smithbob@lnxserver ~]$ ./querydb update.sql
                                  `date -d "-1 year" +%Y`
                                  `date +%Y`
Querying database with SQL code update.sql starting from year 2020 and ending
at year 2021.
```

The number of parameters passed into the script is placed in the variable **`$#`** and can be used to determine if the script is being called with the correct number of parameters:

```
if [ $# -ne 3 ]
then
 echo "SCRIPT CALLED WITH INCORRECT NUMBER OF PARAMETERS...YOU DOLT!"
 exit
fi
```

More Fun with Variables

From time to time, you'll need to create a single variable from a concatenation of several variables in your Linux script. A very clean way to do this is to specify all of the variables to be concatenated together within double-quotes using the alternate form of the resolution syntax for each variable: `${`*`varname`*`}` instead of `$`*`varname`*. For example, let's create the variable `sYYYYMM` from the concatenation of the two individual variables `sYYYY` and `sMM`:

```
sYYYY="2020"
sMM="06"
sYYYYMM="${sYYYY}${sMM}"
echo $sYYYYMM
202006
```

Let's create a variable to hold the four-digit year followed by a dash followed by the two-digit month:

```
sYYYYMM=`date +%Y-%m`
echo $sYYYYMM
2021-12
```

Now, one way to pull out the four-digit year from `$sYYYYMM` is with the `cut` command:

```
echo $sYYYYMM | cut -d'-' -f1
2021

echo $sYYYYMM | cut -d'-' -f2
12
```

`cut`	cut it out!	Retrieves a portion of a delimited text string

The switch `-d` is followed by the desired delimiter, and the switch `-f` is followed by the field number you want to retrieve within the text string. You can set a variable using the code above, like this (using those crazy backticks again!):

```
sYYYY="`echo $sYYYYMM | cut -d'-' -f1`"
echo $sYYYY
2021
```

Now, there's an alternate syntax which I think is a bit cleaner than the backtick-encrusted one shown above, but you can use whichever syntax you prefer:

```
sYYYY="$(cut -d'-' -f1 <<< $sYYYYMM)"
echo $sYYYY
2021
```

Note that the syntax `$(`*command*`)` is equivalent to `` ` ``*command*`` ` ``, but is more highly regarded in wealthier circles. Here, the same `cut` command is being executed, but the variable `sYYYYMM` is being forced into the `cut` command using the `<<<` syntax (Jousting lance? Sergeant stripes? Who the hell knows.).

Now that you know the alternate variable resolution syntax, you can easily convert lowercase to uppercase and vice versa. For example, given the text `taco bell`, to convert to uppercase, use two carets, like this:

```
sTB_LOWER="taco bell"
sTB_UPPER="${sTB_LOWER^^}"
echo $ sTB_UPPER
TACO BELL
```

And, to convert to lowercase, use two commas, like this:

```
sTB_UPPER="TACO BELL"
sTB_LOWER="${sTB_UPPER,,}"
echo $ sTB_LOWER
taco bell
```

Use one caret (**^**) to uppercase the first character, and one comma (**,**) to lowercase the first character.

`${`*var*`^^}`	**UPPERCASE**	Uppercase **entire** string
`${`*var*`^}`	**U**ppercase	Uppercase **first** character
`${`*var*`,,}`	**lowercase**	Lowercase **entire** string
`${`*var*`,}`	**l**OWERCASE	Lowercase **first** character

The `while` Loop and Parameters

Recall that, in the section *Passing Parameters into a Script*, we learned how to grab each parameter individually passed into a script using `$1`, `$2`, etc. and then store those values in separate variables for use later on. This is great if each parameter means something different as in the example in that section: the name of a SQL query file (`$1`), the starting year (`$2`) and the ending year (`$3`). But, if the the parameters are, say, names of database tables to be processed in exactly the same way by the script, you can process each one in turn using a `while` Loop along with the `set` command. This allows for variable numbers of parameters to be passed into the script! Sweet!!

For example, let's create a Linux scripted called `updateTables` which expects a quoted list of blank-delimited table names as parameters (e.g., `"TBL1 TBL2 TBL3"`):

```
#!/bin/bash

# All of the quote-enclosed tables listed on the command line.
TBLLIST="$1"
echo $TBLLIST

# Process each table in turn stored in TBLLIST.
set -- $TBLLIST
while [ $# -gt 0 ]
do

 # Each table name is now known as $1 within the loop.
 sThisTable="$1"

 # Since we grabbed the current iteration`s table name, we
 #  use shift to remove it from the TBLLIST in preparation
 #  for the next loop.
 shift

 echo $sThisTable

done

exit
```

When we run the following from the command line…

```
[smithbob@lnxserver ~]$ ./updateTables "TBL1 TBL2 TBL3 TBL4 TBL5"
```

…we receive the following output:

```
TBL1 TBL2 TBL3 TBL4 TBL5
TBL1
TBL2
TBL3
TBL4
TBL5
```

Now, in order for `$1` to be initially associated with all of the tables listed on the command line for the script, they must be enclosed in quotes, as shown above.

When used with the two dashes, the `set` command manipulates the positional parameters outside the normal way scripts handle positional parameters (`$1`, `$2`, etc.). That is, when used with the `shift` command, the positional parameters slide to the left one by one each time the `shift` command is executed and is the reason `$1` is hard-coded above. That's why each output of the `while` Loop displays the subsequent table name.

Working with Files

Recall that we talked about the `for` loop above and used it to produce a range of numbers with and without a step value. Linux has this wonderful feature which allows you to loop through the **files in a directory** using the `for` loop. This can be a real lifesaver sometimes!! For example, let's assume we have a directory called `sqlfiles` under `/home/smithbob`. In the `sqlfiles` directory, we have five SQL files named `query1.sql` to `query5.sql`. Let's use the `for` loop to print out each file name:

```
for sThisFile in /home/smithbob/sqlfiles/query*.sql
do
 echo "Processing file: $sThisFile."
done
Processing file: /home/smithbob/sqlfiles/query1.sql.
Processing file: /home/smithbob/sqlfiles/query2.sql.
Processing file: /home/smithbob/sqlfiles/query3.sql.
Processing file: /home/smithbob/sqlfiles/query4.sql.
Processing file: /home/smithbob/sqlfiles/query5.sql.
```

Naturally, you can perform a variety of tasks on each file from, say, creating a backup copy, renaming the file, processing the file, etc. rather than just the silly `echo` command. Very nice trick to know, bruh!

Time to Say Buh-Bye!

In our Linux scripts, we've just placed an unassuming `exit` command at the end of the file. But, the `exit` command allows you to specify a return code: a number between `0` and `255` indicating the success-ness or failure-ness of the script. By convention, successfully executed scripts exit with a return code of zero (`0`) while issues occurring during the execution of the script are indicated by a non-zero return code. Note that a non-zero return code doesn't necessarily mean the script failed in a horrible bloody mass of bone and sinew, but just a slight cough. For example, let's exit our script with a return code of zero:

```
#!/bin/bash

# Create our log file.
echo "Program started at `date`." > /home/smithbob/myscript1.log

# Append the contents of /proc/cpuinfo to the log file.
cat /proc/cpuinfo >> /home/smithbob/myscript1.log

# Close out our log file.
echo "Program ended at `date`." >> /home/smithbob/myscript1.log

exit 0
```

Note that if you don't specify a return code, the return code from the **most recently completed command** is used as a proxy. You can programmatically capture each command's return code by using the variable **`$?`** instead. Let's modify the script above to check if the `cat` command is successful:

```
#!/bin/bash

# Create our log file.
echo "Program started at `date`." > /home/smithbob/myscript1.log

# Append the contents of /proc/cpuinfo to the log file.
cat /proc/cpuinfo >> /home/smithbob/myscript1.log
if [[ $? -ne 0 ]]
do
 echo "Something has gone horribly wrong! IT'S SIRENHEAD! RUN AWAY!!"
 exit $?
done

# Close out our log file.
echo "Program ended at `date`." >> /home/smithbob/myscript1.log

exit 0
```

Environment Variables

Have you noticed that, when you execute your scripts, you been specifying `./` before the script name, but other Linux commands don't have that annoying requirement? There are several important variables created when you log into your Linux session which can resolve this minor issue and we're going to chat about them right here, right now, bub!

Linux sets up a series of *environment variables* for you when you log in. These environment variables are very important and can be used within your scripts to make the code a bit more generic. You can easily display a list of environment variables currently defined in your session by executing the `env` command from the Linux command line (below is an abbreviated list of the most adorable environment variables):

```
[smithbob@lnxserver ~]$ env
USERNAME=smithbob
PWD=/home/smithbob
HOME=/home/smithbob
SHELL=/bin/bash
PATH=/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin
JAVA_HOME=/usr/java/latest
CLASSPATH=/usr/lib/hadoop/*:/usr/lib/hadoop/*:.
```

env	**env**ironment	Displays the session environment variables.

The variables shown above, those in capital letters, are a few of the environment variables that may be available in your session when you log into your Linux edge node server. Here's a brief explanation of each:

USERNAME	The username for the current session.
PWD	The current working directory. This environment variable updates as you move around the directory structure using `cd`.
HOME	The user's home directory. This environment variable is very useful for making scripts more generic.
SHELL	The shell that's being used. In this case, the `bash` shell is running. I told you!! I told you!!
PATH	A colon-delimited list of directories that Linux should search through in order to find commands.
JAVA_HOME	The home directory of the Java installation.
CLASS_PATH	A colon-delimited list of directories and/or Java `.jar` files.

Recall that, in the script `myscript1`, we hard-coded the directory `/home/smithbob`. If you're testing a script in your own Linux account, which will eventually be moved over to the generic account, you can avoid having to hard-code this directory by using the `HOME` environment variable instead:

```
#!/bin/bash

# Create our log file.
echo "Program started at `date`." > $HOME/myscript1.log

# Append the contents of /proc/cpuinfo to the log file.
cat /proc/cpuinfo >> $HOME/myscript1.log
if [[ $? -ne 0 ]]
do
 echo "Something has gone wrong! RUN AWAY!!"
 exit $?
done

# Close out our log file.
echo "Program ended at `date`." >> $HOME/myscript1.log

exit 0
```

As you can see in the code above, just like your own script variables, you resolve environment variables by prepending with a dollar-sign `$HOME` or using the alternate syntax `${HOME}`.

Note: When scheduling a script to run at a later date, your environment variables aren't always picked up automatically. One way around this is to force in the all-important .bash_profile file near the top of your script using the source command (which we discussed earlier):

```
#!/bin/bash

# Bring in environment variables.
source $HOME/.bash_profile

# Create our log file.
echo "Program started at `date`." > $HOME/myscript1.log

# Append the contents of /proc/cpuinfo to the log file.
cat /proc/cpuinfo >> $HOME/myscript1.log
if [[ $? -ne 0 ]]
do
 echo "Something has gone wrong! RUN AWAY!!"
 exit $?
done

# Close out our log file.
echo "Program ended at `date`." >> $HOME/myscript1.log

exit 0
```

source	**source**	Brings in information available in the indicated file.

As indicated above, the .bash_profile file contains important information available to your Linux session. Recall that we edited this file earlier to contain the alias lsf as well as an interrogative version of rm. But, you can also alter your environment variables from within this file as well. For example, as indicated above, the PATH environment variable contains a colon-delimited list of directories where executables are stored. When we enter a command at the Linux command line, each one of these directories is searched in turn until the command is found, at which point it's executed. We can modify the PATH environment variable to include the current working directory indicated by the single period (.). In the vi Editor, edit the .bash_profile file (located in the directory /home/smithbob). You'll see a line similar to the following:

```
PATH=$PATH:$HOME/bin
export PATH
```

The first line creates the PATH environment variable by setting to its current definition $PATH and then adds in the bin directory located in $HOME directory. You separate the two with a colon (:). Effectively, this line just adds $HOME/bin to the current PATH environment variable. Now, let's add in the current working directory (.) by simply adding a period as part of the PATH:

```
PATH=$PATH:$HOME/bin:.
export PATH
```

The export command allows the newly updated PATH environment variable to be available outside the cordoned-off confines of the .bash_profile script itself. Don't forget that any changes made to .bash_profile are not immediately available and you still need to tell Linux to execute the .bash_profile script in order for the environment variables to be updated in your session. As we've seen before, simply use the source command:

```
[smithbob@lnxserver ~]$ source .bash_profile
```

Now, when you execute the script myscript1, you no longer need to specify the ./ and can just call the script itself: myscript1. Huzzah!!

Chapter 21 – Running ImpalaSQL from the Linux Command Line

Although we showed some simple examples earlier in the book, in this chapter we'll detail how to run ImpalaSQL queries using the `impala-shell` utility from the Linux command line. We also show you how to pass parameters into an ImpalaSQL query as well as how to resolve those parameters in the SQL code itself. Buckle up! It's gonna be a bumpy ride!

Using `impala-shell`

Despite the plethora of SQL clients you can use to access the Hadoop database and run your fab ImpalaSQL queries, I generally like to use `impala-shell` directly from the Linux command line because some useful Hadoop messages are displayed which are not usually displayed by SQL clients. For example, when running a query in `impala-shell`, you're presented with a URL you can use to view the progress of your query. When using a SQL client, this information is not shown. We talk more about this URL in *Chapter 24 – The Impala Queries Webpages*.

Depending on how your Hadoop Administrator set up the environment, you may simply be able to execute `impala-shell` from the command line without providing any switches to access your database via Impala:

```
[smithbob@lnxserver ~]$ impala-shell
Starting Impala Shell without Kerberos authentication
Connected to hdpserver:21000
Server version: impalad version 2.10.0-cdh5.13.1 RELEASE (build
1e4b23c4eb52dac95c5be6316f49685c41783c51)
***********************************************************************************
Welcome to the Impala shell.
(Impala Shell v2.10.0-cdh5.13.1 (1e4b23c) built on Thu Nov  9 08:29:47 PST 2017)

When pretty-printing is disabled, you can use the '--output_delimiter' flag to set
the delimiter for fields in the same row. The default is ','.
***********************************************************************************
[hdpserver:21000] >
```

Notice that, after the initial messages, you're presented with the `impala-shell` command prompt, shown in bold font above. Just like many other Linux commands, `impala-shell` has several switches you can use to modify its behavior. For example, if your network uses Kerberos, you can add the `-k` switch. If you'd like to specify a specific Hadoop server, you can use the `-i` switch followed by the name of that Hadoop server. And, if you want to start in a specific schema, you can use the `-d` switch followed by the name of the schema. Taken together, here's the extended command:

```
[smithbob@lnxserver ~]$ impala-shell -i hdpserver -d prod_schema
```

`-k`	Access Impala via Kerberos (can use `--kerberos` instead of `-k`).
`-i` *`server_name`*	Access the Hadoop server or *gateway node* specified by *`server_name`* (can use `--impalad=` instead of `-i`).
`-d` *`schema_name`*	Start in the *`schema_name`* schema (can use `--database=` instead of `-d`).
`-h`	Display the `impala-shell` usage page.

If you plan on using `impala-shell` frequently, maybe create an alias for the code above in your `.bash_profile`:

```
alias isps='impala-shell -k -i hdpserver -d prod_schema'
```

Now, you can just type in `isps` at the Linux command line to access the Hadoop database's `prod_schema`.

You can display the usage page for `impala-shell` by specifying the `-h` switch:

```
[smithbob@lnxserver ~]$ impala-shell -h
Usage: impala_shell.py [options]
```

```
Options:
  -h, --help            show this help message and exit
  -i IMPALAD, --impalad=IMPALAD
                        <host:port> of impalad to connect to
                        [default: hdpserver:21000]
  -q QUERY, --query=QUERY
                        Execute a query without the shell [default: none]
  -f QUERY_FILE, --query_file=QUERY_FILE
                        Execute the queries in the query file, delimited by ;.
                        If the argument to -f is "-", then queries are read
                        from stdin and terminated with ctrl-d. [default: none]
  -k, --kerberos        Connect to a kerberized impalad [default: False]
  -o OUTPUT_FILE, --output_file=OUTPUT_FILE
                        If set, query results are written to the given file.
                        Results from multiple semicolon-terminated queries
                        will be appended to the same file [default: none]
  -B, --delimited       Output rows in delimited mode [default: False]
  --print_header        Print column names in delimited mode when pretty-
                        printed. [default: False]
  --output_delimiter=OUTPUT_DELIMITER
                        Field delimiter to use for output in delimited mode
                        [default: \t]
  -s KERBEROS_SERVICE_NAME, --kerberos_service_name=KERBEROS_SERVICE_NAME
                        Service name of a kerberized impalad [default: impala]
  -V, --verbose         Verbose output [default: True]
  -p, --show_profiles   Always display query profiles after execution
                        [default: False]
  --quiet               Disable verbose output [default: False]
  -v, --version         Print version information [default: False]
  -c, --ignore_query_failure
                        Continue on query failure [default: False]
  -r, --refresh_after_connect
                        Refresh Impala catalog after connecting
                        [default: False]
  -d DEFAULT_DB, --database=DEFAULT_DB
                        Issues a use database command on startup
                        [default: none]
  -l, --ldap            Use LDAP to authenticate with Impala. Impala must be
                        configured to allow LDAP authentication.
                        [default: False]
  -u USER, --user=USER  User to authenticate with. [default: smithbob]
  --ssl                 Connect to Impala via SSL-secured connection
                        [default: False]
  --ca_cert=CA_CERT     Full path to certificate file used to authenticate
                        Impala's SSL certificate. May either be a copy of
                        Impala's certificate (for self-signed certs) or the
                        certificate of a trusted third-party CA. If not set,
                        but SSL is enabled, the shell will NOT verify Impala's
                        server certificate [default: none]
  --config_file=CONFIG_FILE
                        Specify the configuration file to load options. File
                        must have case-sensitive '[impala]' header. Specifying
                        this option within a config file will have no effect.
                        Only specify this as a option in the commandline.
                        [default: /home/oracle/.impalarc]
  --live_summary        Print a query summary every 1s while the query is
                        running. [default: False]
  --live_progress       Print a query progress every 1s while the query is
                        running. [default: False]
  --auth_creds_ok_in_clear
```

```
                        If set, LDAP authentication may be used with an
                        insecure connection to Impala. WARNING: Authentication
                        credentials will therefore be sent unencrypted, and
                        may be vulnerable to attack. [default: none]
  --ldap_password_cmd=LDAP_PASSWORD_CMD
                        Shell command to run to retrieve the LDAP password
                        [default: none]
  --var=KEYVAL          Define variable(s) to be used within the Impala
                        session. It must follow the pattern "KEY=VALUE", KEY
                        starts with an alphabetic character and contains
                        alphanumeric characters or underscores. [default:
                        none]
```

Note that the default user shown above for the `-u` switch shows `smithbob`, but this will change depending on which Linux account you're logged in to. For example, if you log into your production account, that account name would appear above instead of `smithbob`.

Once you're at the `impala-shell` command prompt, you can query the database using ImpalaSQL just as you would with a SQL client:

```
[hdpserver:21000] > select * from prod_schema.dim_us_state_mapping;
Query: select * from prod_schema.dim_us_state_mapping
Query submitted at: 2021-12-19 10:54:00 (Coordinator: http://company.com:25000)
Query progress can be monitored at:
http://hdpserver:25000/query_plan?query_id=3141a066db178d37:42ab345f00000000
+------------+----------------------------------------+
| state_code | state_name                             |
+------------+----------------------------------------+
| MH         | MARSHALL ISLANDS                       |
| RI         | RHODE ISLAND                           |
| WI         | WISCONSIN                              |
| WY         | WYOMING                                |
| PA         | PENNSYLVANIA                           |
| CZ         | PANAMA CANAL ZONE                      |
...snip...
```

Take note that you're shown several useful messages such as the submit date and time, the URL to monitor the query's progress, as well as the output from the query. When using a SQL client, these messages won't be shown, but the results of the query, of course, will appear.

As you can see, the output is displayed in a retro teletype style (how quaint!), but if you start `impala-shell` with the `-B` switch, the output will be displayed without the dashes, vertical bars and plus signs, but will instead be delimited based on your specified output delimiter. By default, the output delimiter is a tab, but you can change that by providing the `--output_delimiter` switch to `impala-shell` and specifying your desired delimiter:

```
[smithbob@lnxserver ~]$ impala-shell -i hdpserver -d prod_schema
                                     -B
                                     -output_delimiter=';'
```

```
[hdpserver:21000] > select * from prod_schema.dim_us_state_mapping;
...snip...
MH;MARSHALL ISLANDS
RI;RHODE ISLAND
WI;WISCONSIN
WY;WYOMING
PA;PENNSYLVANIA
CZ;PANAMA CANAL ZONE
...snip...
```

Now, startup messages are shown along with the output, but can be silenced by using the `--quiet` switch. Fantastic!! You can also print the headers as well by providing the `--print_header` switch:

```
[smithbob@lnxserver ~]$ impala-shell -i hdpserver -d prod_schema
                                     -B
                                     -output_delimiter=';'
                                     --quiet
                                     --print_header

[hdpserver:21000] > select * from prod_schema.dim_us_state_mapping;
state_code;state_name
MH;MARSHALL ISLANDS
RI;RHODE ISLAND
WI;WISCONSIN
WY;WYOMING
PA;PENNSYLVANIA
CZ;PANAMA CANAL ZONE
...snip...
```

Note that I wouldn't necessarily create an output file in this manner for queries that produce vast amounts of output, but instead use the `CREATE EXTERNAL TABLE` Statement described in *Chapter 1 – Quick Start Guide*.

Executing ImpalaSQL Queries

In the previous section, we discussed how to submit a SQL query from the `impala-shell` command line. Although nice, it's not very programmy, is it? In this section, we discuss the `impala-shell` switch `-f`, which allows you to submit a file containing one or more SQL queries; and the `-q` switch, which allows you to submit a query in quotes directly from the command line (*très muy nice!*).

Recall, in a previous chapter, that we coded the following in a Linux script using the `-q` switch and passed in a quoted SQL query in quotes:

```
sRowCnt=`impala-shell -B --database=prod_schema
                      -q 'select count(*) from dim_us_state_mapping;'`
echo $sRowCnt
65
```

Now, if you have a file containing one or more SQL queries, you can just run the entire file by specifying the fab `-f` switch followed by the name of the file. For example, let's assume we have the following SQL code in a file named `query1.sql`:

```
use prod_schema;

drop table if exists states_A purge;
create table states_A stored as parquet as
 select state_code,state_name
  from dim_us_state_mapping
  where state_code like 'A%';

drop table if exists states_M purge;
create table states_M stored as parquet as
 select state_code,state_name
  from dim_us_state_mapping
  where state_code like 'M%';
```

Let's run this entire file at the command line using `impala-shell`:

```
[smithbob@lnxserver ~]$ impala-shell -i hdpserver -d prod_schema -f query1.sql
```

Here's the output from this command:

```
Starting Impala Shell without Kerberos authentication
Connected to hdpserver:21000
Server version: impalad version 2.10.0-cdh5.13.1 RELEASE (build
1e4b23c4eb52dac95c5be6316f49685c41783c51)
Query: use prod_schema
Query: drop table if exists states_A purge
Query: create table states_A stored as parquet as
 select state_code,state_name
  from dim_us_state_mapping
  where state_code like 'A%'
Query submitted at: 2021-12-19 14:09:08 (Coordinator: http://hdpserver:25000)
Query progress can be monitored at:
http://hdpserver:25000/query_plan?query_id=35424326c140647d:2acdc4e600000000
+-------------------+
| summary           |
+-------------------+
| Inserted 8 row(s) |
+-------------------+
Fetched 1 row(s) in 0.52s
Query: drop table if exists states_M purge
Query: create table states_M stored as parquet as
 select state_code,state_name
  from dim_us_state_mapping
  where state_code like 'M%'
Query submitted at: 2021-12-19 14:09:08 (Coordinator: http://hdpserver:25000)
Query progress can be monitored at:
http://hdpserver:25000/query_plan?query_id=b44dde19a8b47536:761dede000000000
+--------------------+
| summary            |
+--------------------+
| Inserted 10 row(s) |
+--------------------+
Fetched 1 row(s) in 0.51s
```

Naturally, you can use the redirection arrow (>) to save this output to a log file to peruse later on.

Now, occasionally one or more queries in your file will fail (for other programmers, of course, not you!). If this happens, `impala-shell` will halt at the point of failure. If you'd prefer `impala-shell` to just continue to trudge along processing the remaining queries in the file, provide the switch `-c` (or its doppelganger `--ignore_query_failure`) at the `impala-shell` command line:

```
[smithbob@lnxserver ~]$ impala-shell -c -i hdpserver
                                          -d prod_schema
                                          -f query1.sql
```

Passing Parameters into an ImpalaSQL Query

One very nice feature is the ability to specify parameters on the `impala-shell` command line and then make use of those values within your SQL queries. For example, rather than hard-coding the initial letter of the `state_code`, as in the queries shown above, let's pass in our desired letter instead:

```
[smithbob@lnxserver ~]$ impala-shell -i hdpserver
                                       -d prod_schema
                                       -f query2.sql
                                       --var "stcd=N"
```

The switch `--var` must precede **each parameter** you want to pass into your SQL query. Within quotes, you provide the name of the parameter followed by an equal sign followed by the value associated with that parameter. Here, the parameter `stcd` is being set to `N`. Naturally, our SQL query must be altered to make use of the parameter, so let's assume the file `query2.sql` contains the following code:

```
use prod_schema;

drop table if exists states_${var:stcd} purge;
create table states_${var:stcd} stored as parquet as
 select state_code,state_name
  from dim_us_state_mapping
  where state_code like '${var:stcd}%';
```

Resolving a parameter is similar to using the Linux alternate variable resolution method; that is, you must specify your parameter within **`${}`**. But, in this case, each parameter name must be preceded by the text **`var:`**, as you see in the code above. Here's the output from the `impala-shell` command above:

```
Starting Impala Shell without Kerberos authentication
Connected to hdpserver:21000
Server version: impalad version 2.10.0-cdh5.13.1 RELEASE (build
1e4b23c4eb52dac95c5be6316f49685c41783c51)
Query: use default
Query: drop table if exists states_N purge
Query: create table states_N stored as parquet as
 select state_code,state_name
  from dim_us_state_mapping
  where state_code like 'N%'
Query submitted at: 2021-12-19 14:35:42 (Coordinator: http://hdpserver:25000)
Query progress can be monitored at:
http://hdpserver:25000/query_plan?query_id=1b4c8750e6265d3c:87d66db900000000
+-------------------+
| summary           |
+-------------------+
| Inserted 8 row(s) |
+-------------------+
Fetched 1 row(s) in 0.64s
```

As you can see in the output above, the parameter is being beautifully resolved to **`N`**, as it should. Very nice feature, this! Love it!! I use it frequently to create backups of tables based on a passed in year and month (`yyyymm`) as parameter.

THIS PAGE LEFT INTENTIONALLY BLANK.

THIS ONE WAS AN ACCIDENT.

PART IV - Working with Hadoop

Chapter 22 – Hadoop Commands from Linux (`hadoop/hdfs`)

In *Chapter 1 – Quick Start Guide*, you were exposed to a several Hadoop-specific commands issued from the Linux command line such as `hadoop fs -mkdir`, `hadoop fs -copyFromLocal`, `hadoop fs -ls -R` and `hadoop fs -getmerge`. These commands allow you to talk to the Hadoop Distributed File System (HDFS) directly using commands, for the most part, very similar to the plain ol' Linux commands we learned about in *Chapter 18 – Introduction to the Linux Operating System*.

Now, there are two macho sets of commands you can use to interact with HDFS, one is `hadoop` and the other is `hdfs`. As you see illustrated so deliciously below, the `hdfs` commands are a subset of the `hadoop` commands and primarily operate within HDFS itself, not usually interacting with the local file system. In this book, we'll just stick with the `hadoop` command since (a) we'll often interact with the local file system, and (b) it's the one with the most icing.

Recall that there are two main directory branches of HDFS in our world: the branch swaddling the **managed tables** and the branch swaddling the **external tables**. While you can use many of the `hadoop` commands presented below on managed tables, chances are you don't have permission to do so. And, in any case, it's best to handle your managed tables via SQL DDL (such as `CREATE TABLE`, `DROP TABLE`) rather than with the `hadoop` commands. So, for the remainder of this chapter, we'll deal exclusively with the branch of HDFS swaddling the **external tables**. Please see the response to the Hadoop Administrator E-Mail for the HDFS branch set up for you and your team to use with external tables.

`hadoop` Commands

In this section, we present several useful `hadoop` commands – the ones you'll use most often. We won't go through all of the available `hadoop` commands, since many of them are esoteric and can cause hives. (Get it?? **Hive**…it's a joke‼...*ahem!*…moving on…)

In general, the `hadoop` commands you issue at the Linux command line take on the following format:

```
hadoop fs -command <source> <target>
```

Both *`source`* and *`target`* can specify either the Linux filesystem or HDFS. Take note that *`command`* may be a familiar Linux command, but it must be prepended with a dash. And, if *`command`* requires switches, each must be prepended with a dash (as usual).

If you'd like to list the contents of an HDFS directory, you can use the familiar Linux `ls` command:

```
hadoop fs -ls /data/prod/teams/prod_schema/
```

On the other hand, if you'd like to see all of the directories and files below that, you can use the familiar `-R` switch to perform a recursive `ls`:

```
hadoop fs -ls -R /data/prod/teams/prod_schema/
```

`hadoop fs -ls` *`target`*	Displays the contents of the HDFS directory
`hadoop fs -ls -R` *`target`*	**R**ecursively displays the contents of the HDFS directory

Since working with `hadoop` commands can be a bit nerve racking, if you ever get in trouble, you can display the entire help info:

```
hadoop fs -help
```

Here's part of the enormously vast output displayed to the screen:

```
Usage: hadoop fs [generic options]
        [-appendToFile <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
        [-checksum <src> ...]
        [-chgrp [-R] GROUP PATH...]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-copyFromLocal [-f] [-p] [-l] [-d] [-t <thread count>] <localsrc> ... <dst>]
        [-copyToLocal [-f] [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-count [-q] [-h] [-v] [-t [<storage type>]] [-u] [-x] [-e] [-s] <path> ...]
        [-cp [-f] [-p | -p[topax]] [-d] <src> ... <dst>]
        [-createSnapshot <snapshotDir> [<snapshotName>]]
        [-deleteSnapshot <snapshotDir> <snapshotName>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] [-v] [-x] <path> ...]
        [-expunge [-immediate]]
        [-find <path> ... <expression> ...]
        [-get [-f] [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-getfacl [-R] <path>]
        [-getfattr [-R] {-n name | -d} [-e en] <path>]
        [-getmerge [-nl] [-skip-empty-file] <src> <localdst>]
        [-head <file>]
        [-help [cmd ...]]
        [-ls [-C] [-d] [-h] [-q] [-R] [-t] [-S] [-r] [-u] [-e] [<path> ...]]
        [-mkdir [-p] <path> ...]
        [-moveFromLocal [-f] [-p] [-l] [-d] <localsrc> ... <dst>]
        [-moveToLocal <src> <localdst>]
        [-mv <src> ... <dst>]
        [-put [-f] [-p] [-l] [-d] [-t <thread count>] <localsrc> ... <dst>]
        [-renameSnapshot <snapshotDir> <oldName> <newName>]
        [-rm [-f] [-r|-R] [-skipTrash] [-safely] <src> ...]
        [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
        [-setfacl [-R] [{-b|-k} {-m|-x <acl_spec>} <path>]|[--set <acl_spec> <path>]]
        [-setfattr {-n name [-v value] | -x name} <path>]
        [-setrep [-R] [-w] <rep> <path> ...]
        [-stat [format] <path> ...]
        [-tail [-f] [-s <sleep interval>] <file>]
        [-test -[defsz] <path>]
        [-text [-ignoreCrc] <src> ...]
        [-touch [-a] [-m] [-t TIMESTAMP (yyyyMMdd:HHmmss) ] [-c] <path> ...]
        [-touchz <path> ...]
        [-truncate [-w] <length> <path> ...]
        [-usage [cmd ...]]

-appendToFile <localsrc> ... <dst> :
  Appends the contents of all the given local files to the given dst file. The dst
  file will be created if it does not exist. If <localSrc> is -, then the input is
  read from stdin.
```

```
-cat [-ignoreCrc] <src> ... :
  Fetch all files that match the file pattern <src> and display their content on
  stdout.

...snip...
```

Now, you can tell `help` to focus on a desired command by placing it after `-help`:

```
hadoop fs -help ls

-ls [-C] [-d] [-h] [-q] [-R] [-t] [-S] [-r] [-u] [-e] [<path> ...] :
  List the contents that match the specified file pattern. If path is not
  specified, the contents of /user/<currentUser> will be listed. For a directory a
  list of its direct children is returned (unless -d option is specified).

  Directory entries are of the form:
        permissions - userId groupId sizeOfDirectory(in bytes)
  modificationDate(yyyy-MM-dd HH:mm) directoryName

  and file entries are of the form:
        permissions numberOfReplicas userId groupId sizeOfFile(in bytes)
  modificationDate(yyyy-MM-dd HH:mm) fileName

    -C  Display the paths of files and directories only.
    -d  Directories are listed as plain files.
    -h  Formats the sizes of files in a human-readable fashion
        rather than a number of bytes.
    -q  Print ? instead of non-printable characters.
    -R  Recursively list the contents of directories.
    -t  Sort files by modification time (most recent first).
    -S  Sort files by size.
    -r  Reverse the order of the sort.
    -u  Use time of last access instead of modification for
        display and sorting.
    -e  Display the erasure coding policy of files and directories.
```

If you're a lover of brevity, you can use the `usage` command instead:

```
hadoop fs -usage ls

Usage: hadoop fs [generic options] -ls [-C] [-d] [-h] [-q] [-R] [-t] [-S] [-r] [-u] [-e]
[<path> ...]
```

`hadoop fs -help`	Displays the entire vast help information
`hadoop fs -help` *`command`*	Displays a specific *`command`*'s help information
`hadoop fs -usage` *`command`*	Displays a bit o' info for a specific *`command`*

Recall that the `CREATE EXTERNAL TABLE` Statement expects the `LOCATION` Clause indicating the directory within HDFS where your files are stored. These files will be accessed *en masse* which is why `LOCATION` indicates a **directory** and not a specific **file**. We talk more about working with external tables in *Chapter 23 – Working with Managed and External Tables*. Now, before you can run the `CREATE EXTERNAL TABLE` Statement, the desired location within HDFS must already exist. Just like the Linux command `mkdir`, you can create a directory in HDFS using the same command:

```
hadoop fs -mkdir /data/prod/teams/prod_schema/tmp_postal_code
```

In the code above, we're creating the directory `tmp_postal_code` under the `prod_schema` directory. If you want to remove the empty directory, you can use the `rmdir` command:

```
hadoop fs -rmdir /data/prod/teams/prod_schema/tmp_postal_code
```

If you attempt to remove a directory which is not empty, you'll be greeted with the following message:

```
rmdir: `/data/prod/teams/prod_schema/tmp_postal_code': Directory is not empty
```

You can recursively empty the contents as well as remove the directory itself by using the `rm` command along with the `-R` switch (**PLEASE BE CAREFUL WITH THIS GREAT POWER!**):

```
hadoop fs -rm -R /data/prod/teams/prod_schema/tmp_postal_code
```

An informational message will be displayed indicating that the directory and its contents have been moved to the trash:

```
22/03/05 13:40:56 INFO fs.TrashPolicyDefault: Moved:
'/data/prod/teams/prod_schema/tmp_postal_code' to trash at:
/user/hdfs/.Trash/Current/warehouse/tablespace/external/tmp_postal_code
```

This means that if you've made a mistake, your Hadoop Administrator can restore your directory and its contents. But, be aware that there's limited time until the trash itself is emptied; so, with all speed, please use your fastest and quickest alacrity, instantaneously, here. Now, if you don't want to involve the trash at all, you can specify the `-skipTrash` option and your files will be instantly corpsified:

```
hadoop fs -rm -R -skipTrash /data/prod/teams/prod_schema/tmp_postal_code
```

`hadoop fs -mkdir` *target*	Creates the *target* directory in HDFS
`hadoop fs -rmdir` *target*	Removes the empty *target* directory from HDFS
`hadoop fs -rm -R` *target*	Removes the *target* as well as its contents, placing all in the trash
`hadoop fs -rm -R -skipTrash` *target*	Removes the *target* as well as its contents, skipping the trash

Once again, let's create the `tmp_postal_code` directory:

```
hadoop fs -mkdir /data/prod/teams/prod_schema/tmp_postal_code
```

Since we have this directory, let's copy over our tab-delimited file `dim_postal_code.tsv` from the home directory (`/home/smithbob`) into the `tmp_postal_code` directory in HDFS. To do this, we can use the `hadoop` command `copyFromLocal` providing the local file as the *source* and the HDFS folder as the *target*:

```
hadoop fs -copyFromLocal /home/smithbob/dim_postal_code.tsv
                         /data/prod/teams/prod_schema/tmp_postal_code/
```

To check that the file is actually there, we can use the `ls` command again:

```
hadoop fs -ls -R /data/prod/teams/prod_schema/tmp_postal_code

-rw-r--r--   3 hdfs supergroup    1784376 2022-03-05 13:54
               /data/prod/teams/prod_schema/tmp_postal_code/dim_postal_code.tsv
```

Now, if you want to copy a file from HDFS to a file in the Linux filesystem, you can use the `copyToLocal` command providing the HDFS directory or filename as the *source* and a local directory as the *target*:

```
hadoop fs -copyToLocal /data/prod/teams/prod_schema/tmp_postal_code
                       /home/smithbob/tmp_postal_code
```

Now, the contents of that HDFS directory are available locally, shown below:

```
[smithbob@lnxserver tmp_postal_code]$ lsf
total 1748
drwxr-xr-x.  2 smithbob smithbob      33 Mar  5 14:02 ./
drwx------. 18 smithbob smithbob    4096 Mar  5 14:02 ../
```

```
-rw-r--r--. 1 smithbob smithbob 1784376 Mar  5 14:02 dim_postal_code.tsv
[smithbob@lnxserver tmp_postal_code]$
```

`hadoop fs -copyFromLocal` *source target*	Copies from Linux filesystem *source* to HDFS *target*
`hadoop fs -copyToLocal` *source target*	Copies from HDFS *source* to Linux filesystem *target*

Note that the command `put` is a synonym for `copyFromLocal` and `get` is a synonym for `copyToLocal`.

`hadoop fs -put` *source target*	Copies from Linux filesystem *source* to HDFS *target*
`hadoop fs -get` *source target*	Copies from HDFS *source* to Linux filesystem *target*

Now that we have the file `dim_postal_code.tsv` in the HDFS folder `/data/prod/teams/prod_schema/tmp_postal_code`, we can display some or all of the rows from that file using the familiar Linux commands `cat`, `head` and `tail`. For example, let's try out the `head` command:

```
[smithbob@lnxserver tmp_postal_code]$ hadoop fs -head
               /data/prod/teams/prod_schema/tmp_postal_code/dim_postal_code.tsv
00623   CABO ROJO       PR      18.08643        -67.15222
00633   CAYEY   PR      18.194527       -66.18346699999999
00640   COAMO   PR      18.077197       -66.359104
00676   MOCA    PR      18.37956        -67.08423999999999
00728   PONCE   PR      18.013353       -66.65218
00734   PONCE   PR      17.999499       -66.643934
00735   CEIBA   PR      18.258444       -65.65987
00748   FAJARDO PR      18.326732       -65.652484
00766   VILLALBA        PR      18.126023       -66.48208
00771   LAS PIEDRAS     PR      18.18744        -65.87088
00791   HUMACAO PR      18.147257       -65.82268999999999
00901   SAN JUAN        PR      18.465426       -66.10786
00906   SAN JUAN        PR      18.46454        -66.10079
00909   SAN JUAN        PR      18.442282       -66.06764
00922   SAN JUAN        PR      18.410462       -66.06053300000001
00924   SAN JUAN        PR      18.401917       -66.01194
00961   BAYAMON PR      18.412462       -66.16033
01704   FRAMINGHAM      MA      42.446396       -71.459405
01731   HANSCOM AFB     MA      42.459085       -71.27556
01746   HOLLISTON       MA      42.196065       -71.43797000000001
01749   HUDSON  MA      42.389813       -71.55791000000001
01770   SHERBORN        MA      42.231025       -71.37202000000001
01831   HAVERHILL       MA      42.771095       -71.12205400000001
01856   LOWELL  MA      42.641779       -71.303488
01908   NAHANT  MA      42.427096       -70.92809
01951   NEWBURY MA      4[smithbob@lnxserver tmp_postal_code]$
```

Notice how the `head` command displays more that the usual `10` rows and then abruptly ends, as you can see by the command prompt being located in a wonky place. This is because the `hadoop` command `head` displays **one kilobyte** of data and not `10` rows. Go figure! Similar for the `tail` command:

```
[smithbob@lnxserver tmp_postal_code]$ hadoop fs -tail
               /data/prod/teams/prod_schema/tmp_postal_code/dim_postal_code.tsv
596999999999
51541   HENDERSON       IA      41.137694       -95.39897000000001
67671   VICTORIA        KS      38.861194       -99.15047
30436   LYONS   GA      32.177508       -82.30448
30719   DALTON  GA      34.801861       -84.989796
37013   ANTIOCH TN      36.055115       -86.64782
26537   KINGWOOD        WV      39.472924       -79.69873
57034   HUDSON  SD      43.134318       -96.51958999999999
61259   ILLINOIS CITY   IL      41.369036       -90.9284
```

```
52035   COLESBURG       IA      42.662381       -91.18541
52072   SAINT OLAF      IA      42.927724       -91.38723
22412   FREDERICKSBURG  VA      38.184716       -77.662559
25161   POWELLTON       WV      38.084773       -81.31241
35748   GURLEY  AL      34.710942       -86.38995
31647   SPARKS  GA      31.183567       -83.43559
46374   SAN PIERRE      IN      41.204744       -86.90009000000001
57212   ARLINGTON       SD      44.377534       -97.13878
57236   GARDEN CITY     SD      44.971494       -97.58996
57369   PLATTE  SD      43.435193       -98.89387000000001
39532   BILOXI  MS      30.462388       -88.93293
39730   ABERDEEN        MS      33.833689       -88.55463
28206   CHARLOTTE       NC      35.248292       -80.82747999999999
36033   GEORGIANA       AL      31.655458       -86.76737
37167   SMYRNA  TN      35.968513       -86.52231
62706   SPRINGFIELD     IL      39.79885        -89.65339899999999
52352   WALKER  IA      42.290421       -91.77461
```

Now, if you're a brave soul, you can dump the entire contents of the file using the `cat` command (no one kilobyte limit craziness here, bub!):

```
hadoop fs -cat
            /data/prod/teams/prod_schema/tmp_postal_code/dim_postal_code.tsv
```

I won't display all that data here, but don't forget about the redirection arrow (`>`) or better be quick with *ye olde CTRL-c*, pal!

`hadoop fs -head` *source*	Displays the first one kilobyte of the *source* file
`hadoop fs -tail` *source*	Displays the last one kilobyte of the *source* file
`hadoop fs -cat` *source*	Displays the entire contents of the *source* file

Now, just for shits-and-giggles, let's copy the local file `dim_postal_code.tsv` into the HDFS directory `tmp_postal_code`, but giving it a new name…twice!! Oh, the anticipation!!

```
hadoop fs -copyFromLocal /home/smithbob/dim_postal_code.tsv
              /data/prod/teams/prod_schema/tmp_postal_code/dim_postal_code2.tsv

hadoop fs -copyFromLocal /home/smithbob/dim_postal_code.tsv
              /data/prod/teams/prod_schema/tmp_postal_code/dim_postal_code3.tsv
```

Now, let's check that those files made it there with all of their body parts intact:

```
hadoop fs -ls -R /data/prod/teams/prod_schema/tmp_postal_code

-rw-r--r--   3 hdfs supergroup    1784376 2022-03-05 13:54
               /data/prod/teams/prod_schema/tmp_postal_code/dim_postal_code.tsv
-rw-r--r--   3 hdfs supergroup    1784376 2022-03-05 14:21
              /data/prod/teams/prod_schema/tmp_postal_code/dim_postal_code2.tsv
-rw-r--r--   3 hdfs supergroup    1784376 2022-03-05 14:22
              /data/prod/teams/prod_schema/tmp_postal_code/dim_postal_code3.tsv
```

Good! Now, let's use the `hadoop` command `getmerge` to collect all of the files in the `tmp_postal_code` **directory** into one large **file** in the local filesystem:

```
hadoop fs -getmerge /data/prod/teams/prod_schema/tmp_postal_code
                    /home/smithbob/dim_postal_code_ALL.tsv
```

The original file `dim_postal_code.tsv` contains 43,689 rows. Not surprisingly, the file `dim_postal_code_ALL.tsv` contains $43{,}689 \times 3 = 131{,}067$ rows.

`hadoop fs -getmerge source target_file`	Copies the contents of the files in the HDFS *source* directory into a **single** local filesystem file *target_file*

There are three additional useful commands that return directory information such as the number of files, total size on disk, disk usage, free space and more. For example, let's see how many files and how much space is being sucked up under the HDFS directory `tmp_postal_code`:

```
hadoop fs -count -h /data/prod/teams/prod_schema/tmp_postal_code
1     3     5.1 M /data/prod/teams/prod_schema/tmp_postal_code
```

Note that you can also specify the `-v` switch and the column names will be displayed (which is nice):

```
DIR_COUNT  FILE_COUNT  CONTENT_SIZE  PATHNAME
1          3           5.1 M         .../tmp_postal_code
```

Here's what these four columns mean:

1. `DIR_COUNT` (Directory Count) – the number of directories under `tmp_postal_code`, including itself
2. `FILE_COUNT` (File Count) – the number of files under `tmp_postal_code`
3. `CONTENT_SIZE` (Content Size) – the total number of bytes under `tmp_postal_code` (when using the `-h` switch, as we are here, the size value is *human readable* which is why it's being reported in megabytes)
4. `PATHNAME` (Path Name) – the name of the path associated with the three values above

If you're interested in knowing how much space in total is being used for an entire directory, you can use the `du` command:

```
hadoop fs -du -h -s -v /data/prod/teams/prod_schema/tmp_postal_code

SIZE   DISK_SPACE_CONSUMED_WITH_ALL_REPLICAS  FULL_PATH_NAME
5.1 M  15.3 M                                 .../tmp_postal_code
```

The total size of all three files in the `tmp_postal_code` directory is `5.1M`, as we saw above. The `15.3M` takes into account the number of **replications** of the data. In this case, my standalone Hadoop laptop replicates the files three times. Again, the `-h` switch displays the size in human readable format. The `-s` switch produces a grand total under the directory (as shown above). If you leave off `-s`, all files are displayed individually. The `-v` switch produces the header row.

Finally, if you'd like to know how much space is left in HDFS for you to recklessly use, you can use the `df` command:

```
hadoop fs -df -h

Filesystem                    Size   Used  Available  Use%
hdfs://lnxserver            63.0 G  1.3 G     11.6 G    2%
```

`hadoop fs -count -h source`	Displays the directory count, file count and used space under the HDFS *source* directory
`hadoop fs -du -h -s -v source`	Displays the total space space used and space used taking into account the replication factor under the HDFS *source* directory
`hadoop fs -df -h`	Displays used and available disk space in HDFS

Chapter 23 – Working with Managed and External Tables

Recall in the section labeled *What's the (For)matter, Buddy?* in *Chapter 4 – A Teensy-Weensy Chat about Hadoop*, I used working with an Excel spreadsheet to describe how *input format*, *output format* and *serde* work. As good an explanation as that was, it's not 100% perfect. So, in this chapter I'd like to describe these activities in more detail and how they work with managed and external tables. We'll focus more on working with database tables at the beginning of the chapter, and defer the discussion about how to export tables as delimited text files, described briefly in *Chapter 1 – Quick Start Guide*, until the end of the chapter.

Note that we show some Java code in the discussion below. If you're unfamiliar with Java, please see *Chapter 39 – Quick Start Guide to Java Programming*.

Can We Talk? I Mean, Really, Can We Talk?

In this section, take off your *SQL programmer* sombrero and put on your *general programmer* trilby. If I gave you a file and told you to read in the data, what would you – a fez-wearing general programmer – do? Well, you'd probably initially try to determine if the file was a text file or a binary file. As you know, the two file types require different approaches to *read in the data*. So, for now, let's assume the file is a text file. What would you – a pith-helmet-wearing general programmer – do next? Well, you'd probably open up the file in a your favorite text editor (or Notepad) and try to answer the following questions:

- ☐ What's the field delimiter?
- ☐ Are the text fields enclosed in quotes?
- ☐ What's the escape character?

Now, there are probably more pieces of information you'd jot down such as the format of the date/time columns, the encoding of the text file, the line delimiter, number of header rows, etc., but let's ignore all that hooey for now.

So, what would you – a porkpie-wearing general programmer – do next? Well, you'd use your favorite language to programmatically open the file, as shown in the terribly childish pseudo-code below:

```
oFile = open_the_bloody_file(file=file_location + "/" + file_name,
                             file_type="text",
                             file_mode="read_write");
```

where `file_type` indicates that the file is a text file and `file_mode` indicates that you intend to *read from* as well as *write to* the file. So, you now have an object, `oFile`, which can be used to do stuff with the data in the file. Good! What's next? Well, you'd read in each row of the file one line at a time until there's no more data left in the file, something like this:

```
String sLine;
while(!oFile.AT_END_OF_FILE) {

 //Read in a single line from the text file
 sLine = oFile.read_a_bloody_line();

}
```

In the pseudo-code above, each complete line in the text file is being read into the string variable `sLine`. But, `sLine` is not being processed further in the code above, so let's add pseudo-code to process `sLine`:

```
String sLine = '';
String sSepChar = '\t';
String sQuoteChar = '"';
String sEscapeChar = '\';

//Create a function to blow apart the fields in sLine
```

```
function deserialize(psLine,psSepChar,psQuoteChar,psEscapeChar) {

 String[] asRowData =
                  psLine.split_apart(psSepCharpsQuoteChar,psEscapeChar);

 ...do something useful with the row of data...

}

while(!oFile.AT_END_OF_FILE) {

 //read in a single line from the text file
 sLine = oFile.read_a_bloody_line();

 //Process the line by separating out each field based on the delimiter.
 deserialize(sLine,sSepChar,sQuoteChar,sEscapeChar);

}
```

In the pseudo-code above, we create a function called `deserialize` which is responsible for blowing apart the single line of data into separate fields based on the separator, quote and escape characters. At this point, the function `deserialize` can do something remarkable with the data (What exactly? Who knows or dares to dream!). Take note that the `while` Loop is responsible for punting over each complete line of data from the input text file into the function `deserialize` which then processes it.

Now, back in Hadoop Land, you can think of both `open_the_bloody_file` and the `while` Loop as the *input format* and the function `deserialize` as the *serde*. It's the responsibility of the *input format* to open the file(s) and hand over each row of data to the *serde* which then processes each input row in some way. So, the *input format* must be given some indication as to the file type, such as a text file in this example. But, recall in *Chapter 4 – A Teensy-Weensy Chat about Hadoop*, we indicated that `STORED AS TEXTFILE` is associated with an *input format*, an *output format*, and a *serde* each of which is a Java class. In this case, the *input format* is the Java Class `TextInputFormat` and the *serde* is the Java class `LazySimpleSerDe`. Clearly, `TextInputFormat`, as its name suggests, is primed and ready for some old-fashioned textual luvin'.

But, suppose Mike the Sales Oinker wants you to add more rows of data to the input file. Well, as a fedora-wearing general programmer, how would you do that? You'd append a new row of data to the file ensuring the data is formatted correctly first, something like this:

```
//Create a function to prep a single row of delimited data
function serialize(pasNewData,psSepChar,psQuoteChar,psEscapeChar) {

 String sPrepped_Line =
            pasNewData.join_together(psSepChar,psQuoteChar,psEscapeChar);

 return sPrepped_Line;

}
```

The parameter `pasNewData` is an array of `STRING`s containing a single line of new data. In this case, we created a function called `serialize` which is responsible for prepping the new row of data to be written to the file. The variable `sPrepped_Line` is a delimited text string ready to be appended to an existing, or completely new, file. Taken together, both `serialize` and `deserialize` make up the *serde* and both classes would appear in `LazySimpleSerDe`, `OpenCSVSerde`, or whatever *serde* you're using.

Although we won't show any pseudo-code (you get the point by now), the final step is to use the Java class associated with the *output format* to write `sPrepped_Line` to the file. When specifying `STORED AS TEXTFILE` on the `CREATE TABLE` Statement, the Java class associated with the *output format* is called `HiveIgnoreKey`

`TextOutputFormat`. Taken together, `STORED AS TEXTFILE` confers the following super-powers on your database table:

```
| SerDe Library:                | org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe          | NULL                 |
| InputFormat:                  | org.apache.hadoop.mapred.TextInputFormat                    | NULL                 |
| OutputFormat:                 | org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat  | NULL                 |
```

Note that this conferment is true for both managed tables as well as external tables. Both need to be told how to handle input (*input format*), how to handle output (*output format*) and how to process each individual row of data (*serde*) being handed to it.

As an extreme example, suppose we have a directory in HDFS which contains a series of image files in Portable Network Graphics (PNG) format each of which contains a page of text from, say, a legal document. The goal is to read in each image, use optical character recognition (OCR) to scan the image to pull out the text and then display it in your SQL Client as a `STRING` column. This is very similar to the discussion above, but the *input format* must be able to read in a binary file and hand the image data to the *serde* `deserialize` method which is responsible for running it through a Java OCR method to pull out the text. This text is then displayed, just like data from a `STORED AS TEXTFILE` table. But, how about inserting new data into that table? In this case, the `serialize` method generates a new PNG image file based on the some text provide to it (along with the font family, font size, etc.) and the *output format* would write out the PNG binary data to the HDFS directory as a new `.png` file. In this case, the Java classes associated with the *input format*, *output format* and *serde* must be able to handle this entire *eeevvviiilll* process. And, in this extreme case, you'd need to create your own *input format*, *output format* and *serde* Java classes to get the job done. Of course, this isn't something you really want the database to do and is probably best handled by an underappreciated summer intern.

In both examples above, important pieces of information must be sent into the *serde* in order for things to work properly: in the first example, the separator, quote and escape characters; in the second, the text, font family and font size. These important pieces of information are called *serde properties* and are indicated on the `CREATE TABLE` Statement as `WITH SERDEPROPERTIES` followed by a comma-delimited list of the properties you want to pass to the *serde*. For example, when using the Java `OpenCSVSerde` class, which we describe in more detail further below, you can override the default values for separator, quote and escape characters on the `CREATE TABLE` Statement like this:

```
CREATE EXTERNAL TABLE PROD_SCHEMA.TMP_STATE_DATA(
 STATE_CODE STRING,
 STATE_NAME STRING
)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
WITH SERDEPROPERTIES (
   "separatorChar" = ",",
   "quoteChar"     = "\""
)
STORED AS TEXTFILE
LOCATION '/data/prod/teams/prod_schema/tmp_state_data'
TBLPROPERTIES('skip.header.line.count'='1');
```

Note how the *serde* is being specified on the `ROW FORMAT SERDE` Clause followed by the fully-specified Java class name in quotes. This is in stark contrast to the very simple `STORED AS` *`storage-format`* Clause. The code above overrides two of the serde properties by specifying them on the `WITH SERDEPROPERTIES` Clause. If we take a look at the Java code for the `OpenCSVSerde` class, you'll not only see the `serialize` and `deserialize` methods, but an `initialize` method which pulls in the overriding serde properties (among other things):

```
private char separatorChar;
private char quoteChar;
private char escapeChar;

public static final String SEPARATORCHAR = "separatorChar";
public static final String QUOTECHAR = "quoteChar";
public static final String ESCAPECHAR = "escapeChar";
```

```
@Override
public void initialize(final Configuration conf, final Properties tbl)
                                                    throws SerDeException {

  final List<String> columnNames =
      Arrays.asList(tbl.getProperty(serdeConstants.LIST_COLUMNS).split(","));

  numCols = columnNames.size();

  final List<ObjectInspector> columnOIs = new
                                       ArrayList<ObjectInspector>(numCols);

  for (int i = 0; i < numCols; i++) {
    columnOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
  }

  inspector = ObjectInspectorFactory.getStandardStructObjectInspector(
                                                  columnNames,columnOIs);
  outputFields = new String[numCols];
  row = new ArrayList<String>(numCols);

  for (int i = 0; i < numCols; i++) {
    row.add(null);
  }

  separatorChar = getProperty(tbl,
                              SEPARATORCHAR,
                              CSVWriter.DEFAULT_SEPARATOR);

  quoteChar = getProperty(tbl,
                          QUOTECHAR,
                          CSVWriter.DEFAULT_QUOTE_CHARACTER);

  escapeChar = getProperty(tbl,
                           ESCAPECHAR,
                           CSVWriter.DEFAULT_ESCAPE_CHARACTER);
}
```

Take note of the following lines in the code above:

```
public static final String SEPARATORCHAR = "separatorChar";
public static final String QUOTECHAR = "quoteChar";
public static final String ESCAPECHAR = "escapeChar";
```

When specifying the `WITH SERDEPROPERTIES` Clause on the `CREATE TABLE` Statement, the quoted values, formatted exactly as shown above, are what's expected, as shown in the example below:

```
WITH SERDEPROPERTIES (
   "separatorChar" = ",",
   "quoteChar"     = "\"",
   "escapeChar"    = "\\",
)
```

Also, the three `getProperty` methods above will use the default values stored in the Java `CSVWriter` class if you don't override them with the `WITH SERDEPROPERTIES` Clause. The Java class `CSVWriter` looks, in part, like this:

```
/** The character used for escaping quotes. */
public static final char DEFAULT_ESCAPE_CHARACTER = '"';

/** The default separator to use if none is supplied to the constructor. */
public static final char DEFAULT_SEPARATOR = ',';

/**
 * The default quote character to use if none is supplied to the
 * constructor.
*/
public static final char DEFAULT_QUOTE_CHARACTER = '"';
```

And, as you see, the default separator is a comma, the default quote character is a double-quote, and the default escape character is a double-quote.

Now, in the `CREATE TABLE` Statement above, the `STORED AS TEXTFILE` Clause is specified as well. *'Sup wit' dat?* Since the files being accessed are text files, the usual *input format* and *output format* suspects can be used: `TextInputFormat` and `HiveIgnoreKeyTextOutputFormat`. If there's a need to override the *input format* and *output format*, you can use a modified version of the `STORED AS storage-format` Clause along with the `ROW FORMAT SERDE` Clause, shown below:

```
ROW FORMAT SERDE 'serde-format-Java-classname'
WITH SERDEPROPERTIES ( ... )
STORED AS INPUTFORMAT 'input-format-Java-classname'
          OUTPUTFORMAT 'output-format-Java-classname'
```

Using our PNG discussion as an example, the complete *input format*, *output format* and *serde* **might** look like the following (assuming we took the time to create these Java classes…which we didn't…and we won't…):

```
ROW FORMAT SERDE 'com.company.bob.smith.is.great.PNGSerde'
WITH SERDEPROPERTIES ( 'fontFamily'='Courier', 'fontSize'='10')
STORED AS INPUTFORMAT 'com.company.bob.smith.is.great.PNGInputFormat'
          OUTPUTFORMAT 'com.company.bob.smith.is.great.PNGOutputFormat'
```

Now, based on the discussion above, the `STORED AS storage-format` Clause resolves to the following Java *input format*, *output format* and *serde* classes (more storage formats appear than have been discussed throughout the book):

COMMON STORAGE FORMATS AND THEIR ASSOCIATED JAVA CLASSES		
Storage Format	**Fmt**	**Java Class**
TEXTFILE	S	org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
	I	org.apache.hadoop.mapred.TextInputFormat
	O	org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
PARQUET	S	org.apache.hadoop.hive.ql.io.parquet.serde.ParquetHiveSerDe
	I	org.apache.hadoop.hive.ql.io.parquet.MapredParquetInputFormat
	O	org.apache.hadoop.hive.ql.io.parquet.MapredParquetOutputFormat
KUDU	S	org.apache.hadoop.hive.kudu.KuduSerDe
	I	org.apache.hadoop.hive.kudu.KuduInputFormat
	O	org.apache.hadoop.hive.kudu.KuduOutputFormat
AVRO	S	org.apache.hadoop.hive.serde2.avro.AvroSerDe
	I	org.apache.hadoop.hive.ql.io.avro.AvroContainerInputFormat
	O	org.apache.hadoop.hive.ql.io.avro.AvroContainerOutputFormat
ORC	S	org.apache.hadoop.hive.ql.io.orc.OrcSerde
	I	org.apache.hadoop.hive.ql.io.orc.OrcInputFormat
	O	org.apache.hadoop.hive.ql.io.orc.OrcOutputFormat
SEQUENCEFILE	S	org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe
	I	org.apache.hadoop.mapred.SequenceFileInputFormat
	O	org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat
RCFILE	S	org.apache.hadoop.hive.serde2.columnar.ColumnarSerDe
	I	org.apache.hadoop.hive.ql.io.RCFileInputFormat
	O	org.apache.hadoop.hive.ql.io.RCFileOutputFormat
JSONFILE	S	org.apache.hive.hcatalog.data.JsonSerDe
	I	org.apache.hadoop.mapred.SequenceFileInputFormat

	O	org.apache.hadoop.hive.ql.io.HiveSequenceFileOutputFormat

where `S`=*serde*, `I`=*input format* and `O`=*output format*. Take note that those storage formats dealing with non-text files, such as `PARQUET`, `KUDU`, etc., specify their own *input format* and *output format* to handle the organization of the underlying data file(s) which are, most decidedly, not text.

Note that the Java classes shown above don't appear in one Java `.jar` file, sadly, but across several `.jar` files. If you'd like to see the underlying Java code associated with one of the classes shown above, you should be able to do a quick Google search (Github is usually your best bet here). Also, don't forget that you can download the entire source code for Apache Hadoop, Apache Hive, Apache Impala, Apache Kudu, etc. from their respective websites to see the underlying Java code. Although you may be able to download the appropriate Java `.jar` files, the classes are obfuscated, so you won't be able to see the code, just the class names. ☹

Sounds Great! What's the Bad News, Bub?

Unfortunately, the syntax `ROW FORMAT SERDE 'serde-format-Java-classname'`, `STORED AS INPUTFORMAT 'input-format-Java-classname'` and `STORED AS OUTPUTFORMAT 'output-format-Java-classname'` only appear in HiveQL, not ImpalaSQL. As a reminder, here's the `CREATE TABLE` Statement syntax in ImpalaSQL:

```
CREATE EXTERNAL TABLE database_name.table_name
 (
  column_name_1 data_type_1 COMMENT 'column comment 1',
  column_name_2 data_type_2 COMMENT 'column comment 2',
  ...
  column_name_n data_type_n COMMENT 'column comment n'
 )
 PARTITIONED BY (
                 column_name_p1 data_type_p1 COMMENT 'column comment p1',
                 column_name_p2 data_type_p2 COMMENT 'column comment p2',
                 ...
                 column_name_pk data_type_pk COMMENT 'column comment pk'
                )
 SORT BY (column_name_i, column_name_j, ...)
 COMMENT 'table-comment'
 ROW FORMAT row-format
 WITH SERDEPROPERTIES (
                       'key-1','value-1',
                       'key-2','value-2',
                       ...
                       'key-m','value-m'
                      )
 STORED AS storage-format
 LOCATION 'HDFS-path-to-data-file-directory'
 CACHED IN 'cache-pool-name'
  WITH REPLICATION = replication-value | UNCACHED
 TBLPROPERTIES (
                'key-1','value-1',
                'key-2','value-2',
                ...
                'key-r','value-r'
               )
;
```

In the ImpalaSQL syntax above, `row-format` can take on only the following syntax:

```
DELIMITED
 FIELDS TERMINATED BY 'char'
 ESCAPED BY 'char'
 LINES TERMINATED BY 'char'
```

Note that you cannot specify any Java classes with the syntax shown above. Now, similar to the ImpalaSQL syntax for `row-format` shown directly above, the `row-format` syntax in HiveQL can take on the following syntax:

```
SERDE 'serde-format-Java-classname'
WITH SERDEPROPERTIES ( ... )
```

As you probably guessed already, there's a similar story for `STORED AS storage-format`. In ImpalaSQL, you can only specify the built-in names of the storage formats: `PARQUET`, `TEXTFILE`, `KUDU`, etc. In HiveQL, the `STORED AS` Clause takes either the familiar `STORED AS storage-format` syntax or the following alternate syntax:

```
STORED AS INPUTFORMAT 'input-format-Java-classname'
          OUTPUTFORMAT 'output-format-Java-classname'
```

The moral of the story is: if you want to interact with data other than `PARQUET`, `TEXTFILE`, `KUDU`, etc., then you have to use HiveQL to create the table using the appropriate Java classes based on your type of data. As mentioned earlier in the book, ImpalaSQL cannot interact with all of the storage formats available in HiveQL. So, once the table has been created in HiveQL, you can then create a final table stored, say, as `PARQUET` and then access that table from ImpalaSQL since ImpalaSQL can read `PARQUET` tables.

Going back to my silly PNG example. Once you've created a table by specifying appropriate *input format*, *output format*, and *serde*…

```
ROW FORMAT SERDE 'com.company.bob.smith.is.great.PNGSerde'
WITH SERDEPROPERTIES ( 'fontFamily'='Courier', 'fontSize'='10')
STORED AS INPUTFORMAT 'com.company.bob.smith.is.great.PNGInputFormat'
          OUTPUTFORMAT 'com.company.bob.smith.is.great.PNGOutputFormat'
```

…the data gleaned from the OCR is just text stored as a `STRING`. Create a final table using, say, the `PARQUET` storage format, and then insert the OCR text strings into this final table. At that point, it's just a `PARQUET` table with a column of text which ImpalaSQL can access without a hitch.

We Good-to-Go with Managed and External Tables Then?

For the most part, the discussion up to now really centers around accessing data as an external table. Normally, when working with managed tables, tables are created based on queries from pre-existing tables in the database. This jibes with what we SQL programmers normally do on a day-to-day basis and is a very familiar workflow. But, whether pulling data from the Internet, being e-mailed data from a colleague, snail-mailed a floppy disk from a foreign entity, etc., loading this data is via external tables. If the data is simplistic enough, you can load the data through ImpalaSQL. If the data is more complex, you may need to load the data in HiveQL and create a final table for your team to use from ImpalaSQL.

So, let's have a competition between ImpalaSQL and HiveQL. **LET'S GET READY TO RUMBLE!!** Using the two-letter US state code-to-state name comma-delimited file `us_state_mapping.csv`, which looks like this…

```
state_code,state_name
aa,u.s. armed forces – americas
ae,u.s. armed forces – europe
ak,alaska
al,alabama
ap,u.s. armed forces – pacific
ar,arkansas
```

```
as,american samoa
az,arizona
ca,california
co,colorado
...snip...
```

…we can read this file in using the `CREATE EXTERNAL TABLE` Statement in both HiveQL as well as ImpalaSQL. First, let's copy the file `us_state_mapping.csv` to HDFS in the normal *ever-so-boring-by-now* way:

```
hadoop fs –mkdir /user/hive/warehouse/tmp_us_state_mapping

hadoop fs -copyFromLocal
          /home/smithbob/us_state_mapping.csv
          /user/hive/warehouse/tmp_us_state_mapping/tmp_us_state_mapping.csv

hadoop fs -ls -R /user/hive/warehouse/tmp_us_state_mapping
```

Next, in your SQL Client GUI accessing Impala or via `impala-shell`, we can issue the following ImpalaSQL `CREATE EXTERNAL TABLE` Statement:

```
create external table prod_schema.tmp_us_state_mapping(state_code string,
                                                        state_name string)
 row format
  delimited
   fields terminated by ','
 stored as textfile
 location '/user/hive/warehouse/tmp_us_state_mapping'
 tblproperties('skip.header.line.count'='1');
```

Alternately, in HiveQL we can use the Java class `OpenCSVSerde` to do a similar thing:

```
create external table prod_schema.tmp_us_state_mapping(state_code string,
                                                        state_name string)
 row format serde 'org.apache.hadoop.hive.serde2.OpenCSVSerde'
 with serdeproperties (
    "separatorChar" = ",",
    "quoteChar"     = "\""
 )
 stored as textfile
 location '/user/hive/warehouse/tmp_us_state_mapping'
 tblproperties('skip.header.line.count'='1');
```

In both cases, you can query the external table `prod_schema.tmp_us_state_mapping`:

```
[hdpserver:21000] prod_schema> select *
                             >  from tmp_us_state_mapping;
+------------+----------------------------------------+
| state_code | state_name                             |
+------------+----------------------------------------+
| aa         | u.s. armed forces – americas           |
| ae         | u.s. armed forces – europe             |
...snip...
| wv         | west virginia                          |
| wy         | wyoming                                |
+------------+----------------------------------------+
```

Notice how, in both `CREATE EXTERNAL TABLE` Statements above, the `STORED AS TEXTFILE` Clause is used. The HiveQL code forces the *serde* to use the Java `OpenCSVSerde` class whereas the ImpalaSQL code makes use of the `ROW FORMAT DELIMITED` Clause to indicate the field delimiter. In both cases, the result is the same data,

but different *serde* classes are used to get there: `LazySimpleSerDe` in ImpalaSQL and `OpenCSVSerde` in HiveQL. Since `STORED AS TEXTFILE` was used in both cases, the *input format* is `TextInputFormat` and the *output format* is `HiveIgnoreKeyTextOutputFormat`. Why? Because the input data is a text file and new data inserted will be text as well.

Table Properties and the `TBLPROPERTIES` Clause

You may have noticed I snuck in the `TBLPROPERTIES` Clause in both `CREATE EXTERNAL TABLE` Statements above. This clause allows you to add key/value pairs of information to the table's description using the following syntax:

```
TBLPROPERTIES (
                 'key-1','value-1',
                 'key-2','value-2',
                 ...
                 'key-r','value-r'
                )
```

Although you can add your own key/value pairs, there are a few pre-defined values as well, some of which are listed below:

key	value
`comment`	Specify the table comment here.
`transactional`	If `value` is set to `true`, the table is transactional. If `value` is set to `false`, the table is not transactional.
`external`	If `value` is set to `true`, the table is changed to an external table. If `value` is set to `false`, the table is changed to a managed table.
`external.table.purge`	If `value` is set to `true`, the table and its underlying data are deleted when a `DROP TABLE` Statement is issued on the table. If `value` is set to `false`, the underlying data is not deleted.
`skip.header.line.count`	Specyfing a `value` greater than zero ensures that any header row(s) in the underlying data files are ignored.
`serialization.null.format`	Allows you to specify a character to represent a `NULL` value in `value`.

In the examples above, we specified…

```
tblproperties('skip.header.line.count'='1');
```

…which indicates that the header row (just one line here) appearing in the underlying data file(s) will be ignored. As another example, to specify a single blank to represent a `NULL` value, you can code something like this…

```
tblproperties('serialization.null.format'=' ');
```

…and even…

```
tblproperties('bob.smith.deserves.a.pay.raise'='true');
```

So, Comma-Delimited Text Files, Huh? HOW BORING!!

In this section, I'd like to describe two additional *serde*s which appear in the HiveQL documentation. The first allows you to read in data using a supplied regular expression, `RegexSerDe`. The second allows you to read in JSON formatted data, `JsonSerDe`. Note that additional serdes are available and can found using your best friend, Google. Note that installation of any new Java classes associated with *input format*, *output format* and *serde* may require your hypersonic Hadoop Administrator to be involved.

ADDITIONAL SERDEs AND THEIR ASSOCIATED JAVA CLASSES		
Purpose	**Fmt**	**Java Class**
Read in data using a provided regular expression		
	S	org.apache.hadoop.hive.serde2.RegexSerDe
	I	org.apache.hadoop.mapred.TextInputFormat
	O	org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat
Read in data in JSON format		
	S	org.apache.hadoop.hive.serde2.JsonSerDe
	I	org.apache.hadoop.mapred.TextInputFormat
	O	org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat

Taking a peek at the Java class `RegexSerDe`…

```
public static final String INPUT_REGEX = "input.regex";
public static final String INPUT_REGEX_CASE_SENSITIVE =
                                            "input.regex.case.insensitive";
```

…you'll note that there are two serde properties available:

- ☐ `input.regex` – This serde property accepts a regular expression as its value. Recall in *Chapter 11 – Regular Expressions*, we discussed back references and how they're specified using parentheses. It's the values of the back references which make up the columns in the database table.
- ☐ `input.regex.case.insensitive` – This serde property accepts either `true` or `false` as its value. When set to `true`, the regular expression is considered case **in**sensitive. When set to `false`, the default value, the regular expression is considered case sensitive.

Naturally, both can be specified using the `WITH SERDEPROPERTIES` Clause. For example,

```
row format serde 'org.apache.hadoop.hive.serde2.RegexSerDe'
 with serdeproperties (
    "input.regex" = "...",
    "input.regex.case.insensitive" = "false"
 )
```

For example, given the following simplistic data…

```
"BOB SMITH" 123-45-6780 822-6235 212A
"PEG SMITH" 123-45-6781 822-6236 212B
"JOE SMITH" 123-45-6782 822-6237 212C
"KAT SMITH" 123-45-6783 822-6238 212D
```

…appearing in the file `team_info.txt`, let's use regular expressions to parse the file. Let's copy the file to HDFS:

```
hadoop fs -mkdir /user/hive/warehouse/tmp_team_info

hadoop fs -copyFromLocal
          /home/smithbob/team_info.txt
          /user/hive/warehouse/tmp_team_info/tmp_team_info.txt

hadoop fs -ls -R /user/hive/warehouse/tmp_team_info
```

Next, in HiveQL, let's issue the following `CREATE EXTERNAL TABLE` Statement:

```
CREATE EXTERNAL TABLE tmp_team_info(team_member_name string,
                                    team_member_ss_nbr string,
                                    team_member_phone string,
                                    team_member_office_number string)
 ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
 WITH SERDEPROPERTIES ('input.regex'=
  '^["]{1}(.*)["]{1} (\\d{3}-\\d{2}-\\d{4}) (\\d{3}-\\d{4}) (\\d{3}[A-Z]{1})$')
```

```
STORED AS TEXTFILE
LOCATION '/user/hive/warehouse/tmp_team_info';
```

Take note that we've specified a regular expression as the value of `input.regex`. Note that if the regular expression doesn't match the row, all of the columns will be set to `NULL`. The data stored in the table is below:

```
+------------------+--------------------+-------------------+---------------------------+
| team_member_name | team_member_ss_nbr | team_member_phone | team_member_office_number |
+------------------+--------------------+-------------------+---------------------------+
| BOB SMITH        | 123-45-6780        | 822-6235          | 212A                      |
| PEG SMITH        | 123-45-6781        | 822-6236          | 212B                      |
| JOE SMITH        | 123-45-6782        | 822-6237          | 212C                      |
| KAT SMITH        | 123-45-6783        | 822-6238          | 212D                      |
+------------------+--------------------+-------------------+---------------------------+
```

Using the same data only formatted as JSON (one line per in the file, although it's shown wrapped below):

```
{"team_member_name":"BOB SMITH","team_member_ss_nbr":"123-45-6780","team_member_phone":"822-
6235","team_member_office_number":"212A"}
{"team_member_name":"PEG SMITH","team_member_ss_nbr":"123-45-6781","team_member_phone":"822-
6236","team_member_office_number":"212B"}
{"team_member_name":"JOE SMITH","team_member_ss_nbr":"123-45-6782","team_member_phone":"822-
6237","team_member_office_number":"212C"}
{"team_member_name":"KAT SMITH","team_member_ss_nbr":"123-45-6783","team_member_phone":"822-
6238","team_member_office_number":"212D"}
```

…we can issue a very similar `CREATE EXTERNAL TABLE` Statement using the `JsonSerDe` serde in HiveQL:

```
CREATE EXTERNAL TABLE tmp_team_info_json(team_member_name string,
                                         team_member_ss_nbr string,
                                         team_member_phone string,
                                         team_member_office_number string)
 ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.JsonSerDe'
 STORED AS TEXTFILE
 LOCATION '/user/hive/warehouse/tmp_team_info_json';
```

Note that the column names must match the JSON key names in the file. And the table contains the following data (same as above):

```
+------------------+--------------------+-------------------+---------------------------+
| team_member_name | team_member_ss_nbr | team_member_phone | team_member_office_number |
+------------------+--------------------+-------------------+---------------------------+
| BOB SMITH        | 123-45-6780        | 822-6235          | 212A                      |
| PEG SMITH        | 123-45-6781        | 822-6236          | 212B                      |
| JOE SMITH        | 123-45-6782        | 822-6237          | 212C                      |
| KAT SMITH        | 123-45-6783        | 822-6238          | 212D                      |
+------------------+--------------------+-------------------+---------------------------+
```

Note that the Java class `JsonSerDe` doesn't take any serde properties of note which is why the `WITH SERDEPROPERTIES` Clause doesn't appear in the `CREATE EXTERNAL TABLE` Statement above. It's important to note that the JSON data must appear as one JSON-formatted row per desired database table row.

Exporting a Hadoop Table Into a Delimited Text File

Since we discussed exporting data from Hadoop into a remote database using `sqoop`, it's probably a good idea to discuss how to export tables in Hadoop into delimited text files. Note that we saw an example of this in *Chapter 1 – Quick Start Guide*. In this chapter, we'll expand upon that idea as well as create a Linux script which will perform the same steps on one or more tables.

When creating an external table by specifying `STORED AS TEXTFILE`, the output format will automatically write the data as text. Note that, in some Hadoop flavors, leaving off `STORED AS TEXTFILE` defaults to this anyway, but

you should check your version. This means that anytime you issue an `INSERT` Statement into that external table, no matter how the input table is currently formatted (`PARQUET`, `KUDU`, etc.), the resulting output will be text.
The easiest way to do this is to use the `CREATE EXTERNAL TABLE` Statement along with the `STORED AS TEXTFILE` Clause. It's a good idea to set the table properties `serialization.null.format` to indicate how you want `NULL`s to appear, and set `external.table.purge` to `true` so that when you drop the table, the underlying exported file is removed as well.

Once the table has been created, use the `INSERT` Statement to insert data into the external table. You may also want to take control of how dates/times and other values are formatted in the external table by setting all of the columns to `STRING`s and using the `CAST` or other functions to have the final data formatted exactly as you want.

Finally, once you've finished inserting data into the external table, you can use the Hadoop command `getmerge` from the Linux command line to assemble all of the file pieces of the table into one large final text file.

For example, using ImpalaSQL, let's create a table called `output_textfile` that specifies `STORED AS TEXTFILE` and indicates that the fields are delimited by a tab (`\t`):

```
CREATE EXTERNAL TABLE prod_schema.output_textfile(col1 string,...)
 ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\t'
 STORED AS TEXTFILE
 TBLPROPERTIES('serialization.null.format'='',
               'external.table.purge'='true');
```

Next, let's insert data into this external table:

```
INSERT INTO prod_schema.output_textfile
 SELECT COL1,
        CAST(COL2 AS STRING) AS COL2,
        CAST(COL3 AS STRING FORMAT 'yyyy-mm-dd') AS COL3
 FROM FINAL_DATA_FOR_CLIENT;
```

Before leaving your SQL Client GUI or `impala-shell`, determine the location of the table by issuing

```
desc formatted prod_schema.output_textfile;.
```

and taking note of the `Location` HDFS directory.

From the Linux command line, issue the Hadoop `getmerge` command to assemble all of the file pieces into one large file on the Linux file system in, say, Bob's home directory `/home/smithbob`:

```
hadoop fs -getmerge /data/prod/teams/prod_schema/output_textfile
                    /home/smithbob/output_textfile.tsv
```

Note that you can do something very similar from within HiveQL using the `OpenCSVSerDe` Java class.

The `tableExporter` Linux Script

Based on the information outlined in the previous section, in this section we create a Linux script which automatically exports one or more tables as delimited text files. Since part of the script creates a corresponding `STORED AS TEXTFILE` table based on the table being exported, you may want to ask your diamond-studded Hadoop Administrator to create an additional schema as a target for these exported tables. For example, the schema `prod_schema_export` can be the schema used during the export process from tables located in `prod_schema`. Although the script below drops the table and its underlying data files, the addition of a separate schema will make it easy to find the exported tables and drop those that are no longer needed if the script dies in the middle for some reason.

Note that this script makes use of the tables `prod_schema.all_tables` and `prod_schema.all_tab_columns` which we discuss in *Chapter 33 – Accessing the Hive MetaStore*. These two tables are generated from the Hive MetaStore and contain table names, column names, data types, and so on. These two tables are analogs to Oracle's `ALL_TABLES` and `ALL_TAB_COLUMNS` and similar to metadata tables such as `INFORMATION_SCHEMA.TABLES` and `INFORMATION_SCHEMA.COLUMNS` tables in Microsoft SQL Server, Teradata, and so on. But, you don't have to make use of these cheap-o imitation-o metadata-o tables and can pull directly from the MetaStore database (e.g., MySQL, PostgreSQL, etc.) itself.

The full script `tableExporter` appears below. Note that you must create the directory which stores the delimited files (`/tmp/prod_schema_export`) as well as the log files (`/tmp/prod_schema_export/logs`). Naturally, there's probably a billion ways to do this, but this script combines both Linux and Hadoop commands as well as ImpalaSQL code we've seen before, so I won't go through the code (there are comments throughout). But, be aware that if you're using Hive version 3 or higher, you may be able to access the metadata directly in the `sys` database schema as opposed to our kludged `ALL_TAB_COLUMNS` metadata table.

```
#!/bin/bash -v

#*----------------------------------------------------------------------------*
#* Program:      tableExporter                                                *
#* Author(s):    Bob Smith                                                    *
#* Date:         July 1, 2022                                                 *
#*                                                                            *
#* Application:  Table exporter.                                              *
#*                                                                            *
#* Abstract:     This script exports one or more tables located in the schema *
#*               prod_schema and creates corresponding tables stored as       *
#*               TEXTFILE in the schema prod_schema_export. Delimited text    *
#*               files will be located in /tmp/prod_schema_export.            *
#*                                                                            *
#* Assumptions:  1. Requested tables are located in prod_schema.              *
#*               2. Exported tables are located in prod_schema_export.        *
#*               3. Exported delimited file(s) will be stored in              *
#*                                                   /tmp/prod_schema_export. *
#*               4. Log files are located in /tmp/prod_schema_export/logs.    *
#*                                                                            *
#* Parameters:   1. Delimiter                                                 *
#*               2. E-Mail Address                                            *
#*               3. Tables to export                                          *
#*                                                                            *
#* Input(s):     Hadoop table(s) in prod_schema.                              *
#*                                                                            *
#* Output(s):    Delimited text files.                                        *
#*                                                                            *
#* Example:      ./tableExporter ";"                                          *
#*                "mikethesalesguy@company.com"                               *
#*                "dim_postal_code dim_us_state_mapping"                      *
#*                                                                            *
#* Notes:        1. Must use the external.table.purge property on the CREATE  *
#*               EXTERNAL TABLE SQL code below:                               *
#*                                                                            *
#*                TBLPROPERTIES('external.table.purge'='true')                *
#*                                                                            *
#*               This will allow the DROP TABLE PURGE to completely remove    *
#*               the files as well as the directory from HDFS.                *
#*                                                                            *
#*               2. Ensure EXPORT_DIR exists beforehand!                      *
#*               3. Always boil water in a time of war.                       *
#*                                                                            *
#* Modification History:                                                      *
#* Date          Prog    Mod #   Reason                                       *
#* ---------     ----    -----   -------------------------------------------- *
#*                                                                            *
#*----------------------------------------------------------------------------*

#*----------------------------------------------------------------------------*
#* Check that the number of incoming arguments is correct.                    *
#*----------------------------------------------------------------------------*
if [ "$#" != "3" ]
 then

  echo "tableExporter: Not enough parameters provided on the command line."
```

```
    echo ""
    echo "Syntax:"
    echo " tableExporter  delimiter-in-quotes  email-address-in-quotes
                                                           space-delimited-list-of-tables-in-quotes"
    echo ""
    echo "Note: Exported files are located in the directory /tmp/prod_schema_export."
    echo "      Log files are located in the directory /tmp/prod_schema_export/logs."

fi

#*---------------------------------------------------------------------------*
#* Initialize variables used throughout the script.                          *
#*---------------------------------------------------------------------------*
#* Directory where the delimited files will be stored *
EXPORTDIR="/tmp/prod_schema_export"
echo "EXPORT DIRECTORY: $EXPORTDIR"

#* Directory where the log files will be placed *
LOGDIR="/tmp/prod_schema_export/logs"
echo "LOG DIRECTORY: $LOGDIR"

#* Argument count *
ARGCNT=$#
echo "NUMBER OF ARGUMENTS: $ARGCNT"

#* Requested delimiter *
DLM=$1
echo "DELIMITER: $DLM"

#* E-Mail address *
EMAIL=$2
echo "E-MAIL: $EMAIL"

#* List of tables to export *
TBLLIST=$3
echo "TABLES TO EXPORT: $TBLLIST"

#* Create input schema *
INSCHEMA="prod_schema"
echo "INPUT SCHEMA: $INSCHEMA"

#* Create output schema *
OUTSCHEMA="prod_schema_export"
TARGETDB=$OUTSCHEMA
echo "OUTPUT SCHEMA: $OUTSCHEMA"
echo "TARGET DB: $TARGETDB.db"

#* Create log file directory/name *
DT="`date +%Y%m%d%H%M`"
LOGDIR="$EXPORTDIR/logs"
LOGFILE="$LOGDIR/$DT.log"
echo "LOG FILE: $LOGFILE"

#*---------------------------------------------------------------------------*
#* Produce some nice looking output to justify our enormous salary.          *
#*---------------------------------------------------------------------------*
echo "Table Exporter" > $LOGFILE
echo " Run Date/Time: `date`" >> $LOGFILE
echo " Delimiter: $DLM" >> $LOGFILE
echo " E-Mail: $EMAIL" >> $LOGFILE
echo " Source DB: $INSCHEMA" >> $LOGFILE
echo " Target DB: $TARGETDB" >> $LOGFILE
echo " Export Table List: $TBLLIST" >> $LOGFILE

#*---------------------------------------------------------------------------*
#* Create upper- and lowercased versions of the INSCHEMA.                    *
#*---------------------------------------------------------------------------*
DB_LC=${INSCHEMA,,}
DB_UC=${INSCHEMA^^}

#*---------------------------------------------------------------------------*
#* Create a subdirectory which will hold the delimited files.                *
#* Note: You could supplement the code with a unique project number or other  *
#*       identifier in order to keep the files separated.                    *
#* Since both /tmp/prod_schema_export/logs and /tmp/prod_schema_export have   *
```

```
#*  been created up-front, no need to run the code below.                   *
#*-------------------------------------------------------------------------*
#CREATESUBDIR="mkdir $EXPORTDIR"
#echo " Output Subdirectory: $CREATESUBDIR" >> $LOGFILE
#eval $CREATESUBDIR

#*-------------------------------------------------------------------------*
#* Loop through the list of requested tables processing each one at a time.   *
#*-------------------------------------------------------------------------*
set -- $TBLLIST
while [ $# -gt 0 ]
do
 #*-------------------------------------------------------------------------*
 #* Pull in the first table as well as create upper- and lowercased versions.  *
 #*-------------------------------------------------------------------------*
 TBL="$1"
 TBL_LC=${TBL,,}
 TBL_UC=${TBL^^}
 echo "---------------------------------------------------------------------------------" >> $LOGFILE
 echo " Exporting the following table: $TBL" >> $LOGFILE
 shift

 #*-------------------------------------------------------------------------*
 #* Create SQL to pull in the column names/data types for the current table.   *
 #* Execute this code using impala-shell and the -q switch.                    *
 #*-------------------------------------------------------------------------*
 SQL_TBLDEFN="
             SELECT LOWER(ALL_COL_INFO) AS ALL_COLL_INFO
              FROM (
                    SELECT TABLE_NAME,GROUP_CONCAT(COL_INFO,',') AS ALL_COL_INFO
                     FROM (
                           SELECT TABLE_NAME,CONCAT_WS(' ',TRIM(COLUMN_NAME),TRIM(DATA_TYPE))
                                                                                          AS COL_INFO
                            FROM (
                                  SELECT TABLE_NAME,COLUMN_NAME,DATA_TYPE
                                   FROM ALL_TAB_COLUMNS
                                   WHERE UPPER(DATABASE_NAME)='$DB_UC'
                                         AND UPPER(TABLE_NAME)='$TBL_UC'
                                   ORDER BY TABLE_NAME,COLUMN_ID
                                   LIMIT 1000000
                                 ) A
                          ) B
                     GROUP BY TABLE_NAME
                   ) C;
            "
 echo " SQL_TBLDEFN: $SQL_TBLDEFN" >> $LOGFILE
 COLDEFN=`impala-shell -i lnxserver --database=$DB_UC -B --quiet -q "$SQL_TBLDEFN" 2>/dev/null`
 echo " COLDEFN: $COLDEFN" >> $LOGFILE

 #*-------------------------------------------------------------------------*
 #* Create similar SQL to the above, but for use with the INSERT Statement.    *
 #* Execute this code using impala-shell and the -q switch.                    *
 #*-------------------------------------------------------------------------*
 SQL_TBLDEFN_INSERT="
                    SELECT LOWER(ALL_COL_INFO) AS ALL_COLL_INFO
                     FROM (
                           SELECT TABLE_NAME,GROUP_CONCAT(COL_INFO,',') AS ALL_COL_INFO
                            FROM (
                                  SELECT TABLE_NAME,TRIM(COLUMN_NAME) AS COL_INFO
                                   FROM (
                                         SELECT TABLE_NAME,COLUMN_NAME
                                          FROM ALL_TAB_COLUMNS
                                          WHERE UPPER(DATABASE_NAME)='$DB_UC'
                                                AND UPPER(TABLE_NAME)='$TBL_UC'
                                          ORDER BY TABLE_NAME,COLUMN_ID
                                          LIMIT 1000000
                                         ) A
                                 ) B
                            GROUP BY TABLE_NAME
                          ) C;
                   "
 echo " SQL_TBLDEFN_INSERT: $SQL_TBLDEFN_INSERT" >> $LOGFILE
 COLDEFN_INSERT=`impala-shell -i lnxserver --database=$DB_UC -B --quiet -q "$SQL_TBLDEFN_INSERT"
                                                                                         2>/dev/null`
 echo " COLDEFN_INSERT: $COLDEFN_INSERT" >> $LOGFILE
```

```
#*-------------------------------------------------------------------------------*
#* Drop the external table if it already exists.                                 *
#*-------------------------------------------------------------------------------*
SQL_DROP="DROP TABLE IF EXISTS $TARGETDB.$TBL_UC PURGE;"
echo " SQL_DROP: $SQL_DROP" >> $LOGFILE
impala-shell -i lnxserver --database=$DB_UC -B --quiet -q "$SQL_DROP" >> $LOGFILE 2>&1

#*-------------------------------------------------------------------------------*
#* Create the external table for this iteration's table.                         *
#*-------------------------------------------------------------------------------*
SQL_CREATE_EXTERNAL_TABLE="
                           CREATE EXTERNAL TABLE $TARGETDB.$TBL_LC($COLDEFN)
                            ROW FORMAT DELIMITED
                            FIELDS TERMINATED BY '$DLM'
                            STORED AS TEXTFILE
                            TBLPROPERTIES('serialization.null.format'=' ',
                                          'external.table.purge'='true');
                          "
echo " SQL_CREATE_EXTERNAL_TABLE: $SQL_CREATE_EXTERNAL_TABLE" >> $LOGFILE
impala-shell -i lnxserver --database=$DB_UC -B --quiet -q "$SQL_CREATE_EXTERNAL_TABLE" >> $LOGFILE
                                                                                                    2>&1

#*-------------------------------------------------------------------------------*
#* Insert data into the external table.                                          *
#*-------------------------------------------------------------------------------*
SQL_INSERT="
            INSERT INTO $TARGETDB.$TBL_UC
             SELECT $COLDEFN_INSERT
              FROM $DB_UC.$TBL_UC;
           "
echo " SQL_INSERT: $SQL_INSERT" >> $LOGFILE
impala-shell -i lnxserver --database=$DB_UC -B --quiet -q "$SQL_INSERT" >> $LOGFILE 2>&1

#*-------------------------------------------------------------------------------*
#* Create the delimited text file from the external table files in HDFS.         *
#*-------------------------------------------------------------------------------*
CREATE_TEXT_FILE="hadoop fs -getmerge
                     hdfs://lnxserver.com:8020/warehouse/tablespace/external/hive/$TARGETDB.db/$TBL_LC
                     $EXPORTDIR/$TBL_LC.txt"
eval $CREATE_TEXT_FILE

#*-------------------------------------------------------------------------------*
#* Generate the headers for the text file.                                       *
#*-------------------------------------------------------------------------------*
SQL_HEADER="
            SELECT LOWER(ALL_COL_HEADER) AS ALL_COL_HEADER
             FROM (
                   SELECT TABLE_NAME,GROUP_CONCAT(COL_INFO,'$DLM') AS ALL_COL_HEADER
                    FROM (
                          SELECT TABLE_NAME,CONCAT_WS(' ',TRIM(COLUMN_NAME)) AS COL_INFO
                           FROM (
                                 SELECT TABLE_NAME,COLUMN_NAME
                                  FROM ALL_TAB_COLUMNS
                                  WHERE UPPER(DATABASE_NAME)='$DB_UC'
                                        AND UPPER(TABLE_NAME)='$TBL_UC'
                                  ORDER BY TABLE_NAME,COLUMN_ID
                                  LIMIT 1000000
                                ) A
                         ) B
                    GROUP BY TABLE_NAME
                  ) C;
           "
COLHDR=`impala-shell -i lnxserver --database=$DB_UC -B --quiet -q "$SQL_HEADER" 2>/dev/null`
echo " Column Headers: $COLHDR" >> $LOGFILE

#*-------------------------------------------------------------------------------*
#* Create the final text file with the column headers on line one using sed.     *
#*-------------------------------------------------------------------------------*
TBL_FINAL_0="sed -e '1i\\"
TBL_FINAL="$TBL_FINAL_0$COLHDR' $EXPORTDIR/$TBL_LC.txt > $EXPORTDIR/$TBL_LC.dlm"
echo " TBL_FINAL: $TBL_FINAL" >> $LOGFILE
eval $TBL_FINAL

#*-------------------------------------------------------------------------------*
```

```
  #* Remove unneeded files.                                                 *
  #*------------------------------------------------------------------------*
  rm -f $EXPORTDIR/$TBL_LC.txt
  rm -f $EXPORTDIR/.$TBL_LC.txt.crc

done

#*------------------------------------------------------------------------*
#* Count the  rows in each exported table as a sanity check for the user.     *
#*------------------------------------------------------------------------*
echo " Row Counts: " >> $LOGFILE
wc -l $EXPORTDIR/*.dlm >> $LOGFILE
echo "" >> $LOGFILE

#*------------------------------------------------------------------------*
#* Produce a useful message for the user.                                 *
#*------------------------------------------------------------------------*
echo " Your table(s) have been exported and are located in $EXPORTDIR." >> $LOGFILE
echo " After FTP'ing your files over, please delete them from the server tout de suite!" >> $LOGFILE

#*------------------------------------------------------------------------*
#* E-Mail the user as well as the programmer of this script.              *
#*------------------------------------------------------------------------*
cat $LOGFILE | mail -s "Export Complete" $EMAIL
cat $LOGFILE | mail -s "Export Complete" smithbob@company.com

exit
```

For example, at the Linux command line, you can run the script above like this:

```
[smithbob@lnxserver ~]$ ./tableExporter ";"
                                        "mikethesalesguy@company.com"
                                        "dim_postal_code dim_us_state_mapping"
```

Once the script completes, you can check out the delimited files in `/tmp/prod_schema_export`:

```
[smithbob@lnxserver test]$ cd /tmp/prod_schema_export/
[smithbob@lnxserver prod_schema_export]$ lsf
total 1828
drwxrwxr-x.   3 smithbob smithbob      77 Apr 24 10:46 ./
drwxrwxrwt. 600 root   root     61440 Apr 24 10:48 ../
-rw-rw-r--.   1 smithbob smithbob 1784423 Apr 24 10:46 dim_postal_code.dlm
-rw-rw-r--.   1 smithbob smithbob    1011 Apr 24 10:46 dim_us_state_mapping.dlm
drwxrwxr-x.   2 smithbob smithbob      30 Apr 24 10:46 logs/
[smithbob@lnxserver prod_schema_export]$
```

Note that at the end of the log file, a list of exported tables along with their row counts appears as a double-check. Note that each table's row count is one higher than the total number of rows due to the addition of the fab header row.

```
Row Counts:
  43690 /tmp/prod_schema_export/dim_postal_code.dlm
     66 /tmp/prod_schema_export/dim_us_state_mapping.dlm
  43756 total
```

Finally, the recommendation to use `/tmp` as your data playground is just for testing. It's best to either ask your Linux Administrator to create a separate directory for you, or create a directory under the location created for you and your team (as indicated in the Hadoop Administrator E-Mail).

Chapter 24 – The Impala Queries Webpages

Recall I mentioned that, occasionally, you may need to kill one or more of your stubborn ImpalaSQL queries. In many cases, this can easily be done from within the SQL client you're using. For example, if you're running a query from `impala-shell`, you can try hitting `CTRL+c` to kill the query. If you're using Toad Data Point, SQuirreL, or other nifty GUI SQL client, try clicking the *stop* or *cancel* or *halten sie* button. But, this doesn't always work and surface-to-air missiles must be employed. Luckily, Impala creates one or more webpages listing all of the currently running (or *in flight*) queries, as well as several recently finished (or *completed*) queries. Now, there are two ways to determine the URL for the query you want to kill. Leaving out the useless Venn Diagram, the two ways are…

1. …when you're given the URL…
2. …when you're not given the URL…

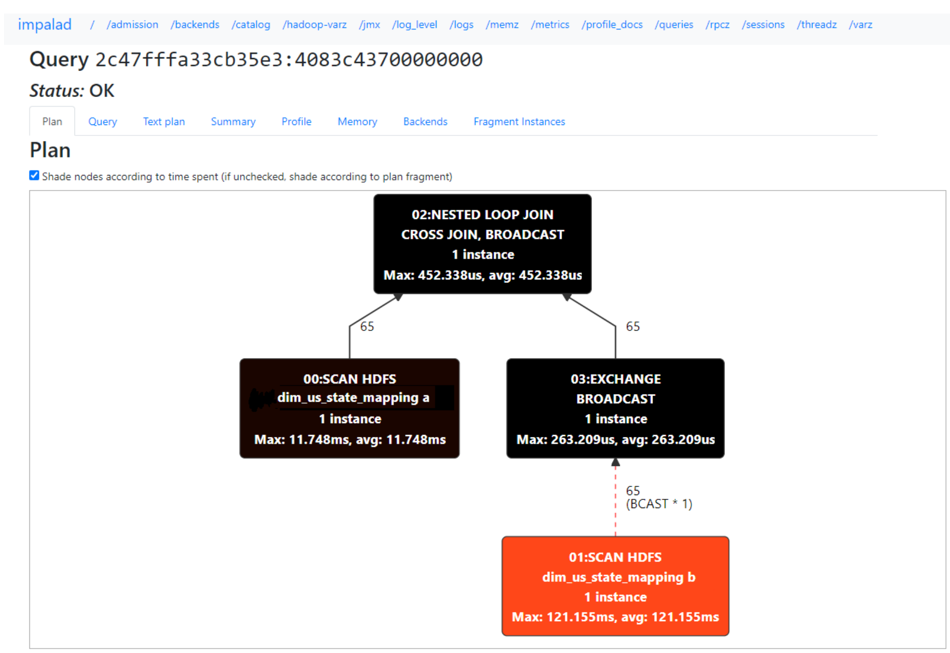
Lemme 'splain.

When You're Given the URL

If you're using `impala-shell`, you'll be shown the URL of the submitted query at the command line. For example,

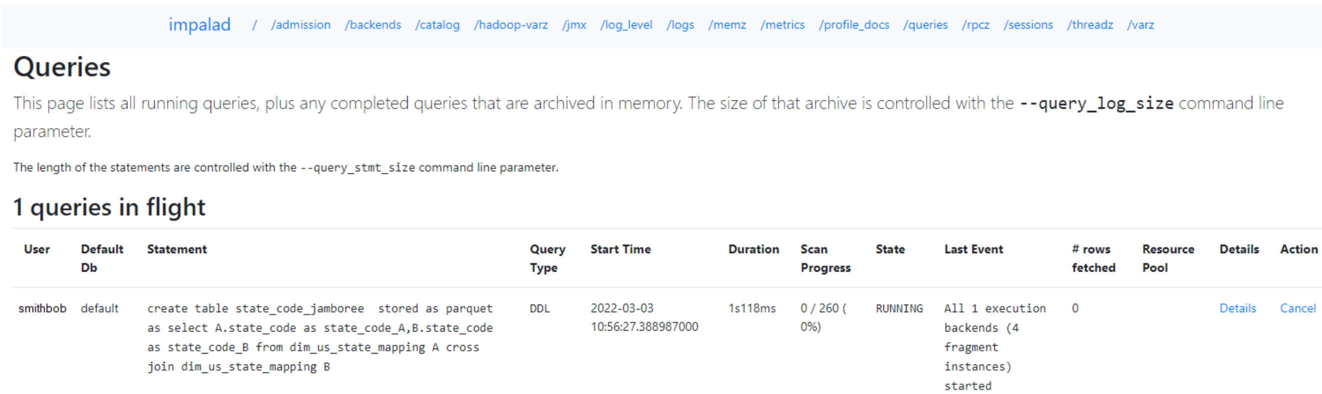
```
[hdpserver.com:21000] prod_schema> create table state_code_jamboree stored as
parquet as select A.state_code as state_code_A,B.state_code as state_code_B
from dim_us_state_mapping A cross join dim_us_state_mapping B;

Query progress can be monitored at:
http://lnxserver:25000/query_plan?query_id=2c47fffa33cb35e3:4083c43700000000

+----------------------+
| summary              |
+----------------------+
| Inserted 4225 row(s) |
+----------------------+
Fetched 1 row(s) in 0.52s
```

The URL following the text "`Query progress can be monitored at:`" is the URL you place in your browser to see the status of the SQL query, whether *in flight* or *completed*. For example, below is an image of the completed query above:



Note, again, that this page displays information about this particular query. In the image above, you see a graphical representation of your query along with the number of rows processed, `65` in our case. These row counts will automatically increase until all of the rows have been processed for that portion of the query. At that point, the *in flight* query will be a *completed* query. Depending on the complexity of your query, some portions of the plan may complete early and will patiently wait until other portions are finished executing to continue on.

If you'd like to see all of the running queries, click the `/queries` link at the top of the page. In this case, you'll see something like this:


impalad / /admission /backends /catalog /hadoop-varz /jmx /log_level /logs /memz /metrics /profile_docs /queries /rpcz /sessions /threadz /varz

Queries

This page lists all running queries, plus any completed queries that are archived in memory. The size of that archive is controlled with the **--query_log_size** command line parameter.

The length of the statements are controlled with the --query_stmt_size command line parameter.

1 queries in flight

User	Default Db	Statement	Query Type	Start Time	Duration	Scan Progress	State	Last Event	# rows fetched	Resource Pool	Details	Action
smithbob	default	create table state_code_jamboree stored as parquet as select A.state_code as state_code_A,B.state_code as state_code_B from dim_us_state_mapping A cross join dim_us_state_mapping B	DDL	2022-03-03 10:56:27.388987000	1s118ms	0 / 260 (0%)	RUNNING	All 1 execution backends (4 fragment instances) started	0		Details	Cancel

Although I've isolated just the one query, this page will display all of the *in flight* and *completed* queries on that server. Note that, while a query is still running, you'll see a `Cancel` link under the heading **Action** at the far right of the webpage. If you click this link, the query will be killed. At this point, whatever SQL client you're using should return…it'll feel dejected…but, it'll return. Clicking the `Details` link will display the query-specific page.

When You're Not Given the URL

Occasionally, `impala-shell` will not display an URL. But, `impala-shell` isn't alone in this since most GUI SQL clients won't display it either. Recall that one of the questions in the Hadoop Administrator E-Mail (see *Chapter 2 – Hadoop Administrator E-Mail*) asked for a list of the Hadoop query webpage URLs. Now, if you're running the standalone version of Hadoop, as I am, there's only one Hadoop query webpage to worry about, but the multinational, multibillion dollar corporation you work for will have at least two, maybe even three! Dare to dream!!

In any case, your query could appear on any one of those webpages and hunting through each page one at a time can be a pain. Please provide your department's web programmer the Hadoop query webpage URLs (see *Chapter 2 – Hadoop Administrator E-Mail*) and ask to have them all placed on one large department webpage. Once completed, your team need only go to that one department-specific webpage to scan for and kill a recalcitrant query. Simples!!

PART V - HPL/SQL Procedural Language

Chapter 25 – Introduction to HPL/SQL

In this chapter, we introduce HPL/SQL, a procedural language similar to Oracle's PL/SQL and Microsoft SQL Server's T-SQL, among others. When ImpalaSQL just doesn't cut it, HPL/SQL will be your best bud! Now, unlike PL/SQL and T-SQL, HPL/SQL is not natively available to run from **within** the database itself. HPL/SQL programs need to be run from the **Linux command line** using the `hplsql` utility. So, you may need to break apart your legacy procedure code into pieces: the first piece runs some SQL code, the second piece submits some HPL/SQL programs, the third piece runs some more SQL code, and so on. You get the general idea.

The developers did a fantastic job creating HPL/SQL! It combines the syntax from several procedural languages, so whether you're coming from an Oracle, SQL Server, IBM DB2, Teradata, PostgreSQL, MySQL, Netezza, etc. background, HPL/SQL may be easier to learn than you think.

Now, HPL/SQL allows you to connect to Impala, Hive, MySQL and many other databases in order to interact with them. HPL/SQL runs alongside the all-important `hplsql-site.xml` file which contains information allowing for connections to disparate databases. In this book, we'll stick mostly with ImpalaSQL (although we do talk a bit about MySQL when we discuss the MetaStore in *Chapter 33 – Accessing the Hive MetaStore*). Just be aware that you can connect to a variety of databases as long as the `hplsql-site.xml` file contains the appropriate JDBC connection information and the associated Java JDBC files are accessible. We talk about the `hplsql-site.xml` file in more detail below.

Finally, if you and your Hadoop Administrator are having trouble getting the `hplsql` utility to execute properly, please see *Appendage #3 – When HPL/SQL Causes You Pain* for some useful information.

`hplsql` Command Line Utility

In order to execute an HPL/SQL program, you use the command line utility `hplsql` and provide it the name of your HPL/SQL program file as well as pass it one or more parameters, if necessary. The basic syntax is:

```
hplsql -f hplsql_program_file.hplsql --define parm1='value1'
                                     --define parm2='value2'
                                     ...
                                     --define parmN='valueN'
```

For example, given an HPL/SQL file called `integrate_x2.hplsql`, which integrates x^2 between the two points $x0$ and $x1$ using the rectangular approximation method, you can run the code like this (Captain OCD says, "Place everything on one line, kids!"):

```
[smithbob@lnxserver ~]$ hplsql -f integrate_x2.hplsql --define pX0=0
                                                      --define pX1=10
                                                      --define pN=1000000
```

Note that the keyword `--define` must appear before each parameter, as shown above. Instead of the `-f` switch, you can use the `-e` switch to pass in some code from the command line, similar to how `impala-shell` accepts the `-q` switch:

```
hplsql -e '...some SQL code...' --define parm1='value1'
```

Note that you should place quotes around parameters containing text, but it's not necessary for numeric parameters.

For example, to test the built-in HPL/SQL substring function `SUBSTR()`, you can run the following:

```
[smithbob@lnxserver ~]$ hplsql -e "SUBSTR('ABC',x0,x1)" --define x0=1
                                                        --define x1=2
```

The output is something like the following, with the results appearing after the annoying messages:

```
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/opt/cloudera/parcels/CDH-7.1.7-
1.cdh7.1.7.p0.15945976/jars/log4j-slf4j-impl-2.13.3.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/opt/cloudera/parcels/CDH-7.1.7-
1.cdh7.1.7.p0.15945976/jars/slf4j-log4j12-1.7.30.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
WARNING: Use "yarn jar" to launch YARN applications.
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/opt/cloudera/parcels/CDH-7.1.7-
1.cdh7.1.7.p0.15945976/jars/log4j-slf4j-impl-2.13.3.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/opt/cloudera/parcels/CDH-7.1.7-
1.cdh7.1.7.p0.15945976/jars/slf4j-log4j12-1.7.30.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.apache.logging.slf4j.Log4jLoggerFactory]
AB
```

Results are here!!

hplsql **-f** *input_file*	Executes the *input_file*
hplsql **-e '**...*some code*...**'**	Executes the *code* within single or double quotes
hplsql ... **--define** *parm1=value1...*	Passes in one or more parameters to the HPL/SQL code

Two additional useful switches are --trace and --version:

hplsql ... **--trace**	Executes the HPL/SQL code but provides additional debugging info
hplsql **--version**	Displays the HPL/SQL version number

When passing parameters with the --define switch, it's perfectly acceptable to specify an empty string as a value:

```
hplsql -f hplsql_program_file.hplsql --define parm1=''
```

We discuss how to resolve parameters within an HPL/SQL program later on.

And, as I've mentioned *ad nauseum*, don't forget that you can make use of the backticks (``) as well as the date utility when running your HPL/SQL program from the command line to specify a current date as well as a shifted date:

```
[smithbob@lnxserver ~]$ hplsql -f pgm1.hplsql
                                --define beg_yyyymm=`date -d "-11 month" +%Y%m`
                                --define end_yyyymm=`date +%Y%m`
```

Introducing the All-Important hplsql-site.xml File

Before we discuss HPL/SQL syntax, we must have a heart-to-heart about the hplsql-site.xml file. This file contains connection information allowing you to connect to, and interact with, a database. Without a properly updated hplsql-site.xml file, you won't be able to interact with, say, Impala, Hive, MySQL, Oracle, etc. The hplsql-site.xml file is made up of one large Configuration Section (enclosed by the start tag <configuration> and end tag </configuration>). The Configuration Section itself contains one or more Property Sections (enclosed by the start tag <property> and end tag </property>), like this:

```
<configuration>
 <property>
 </property>
</configuration>
```

Each Property Section, defines a connection to a specific database and contains the following XML tags:

- ☐ <name>...</name> - A name given to the connection used within an HPL/SQL program to connect to that specific database
- ☐ <value>...</value> - The JDBC connection description for this database
- ☐ <description>...</description> - A description of the connection

For example, the following Property Section defines a connection to **Impala** starting in the `default` database:

```
<configuration>
 <property>
  <name>hplsql.conn.impala</name>
  <value>com.cloudera.impala.jdbc4.Driver;jdbc:impala://hdpserver:21050/
                                          default;smithbob;PaSsWoRd123;</value>
  <description>Impala JDBC Connection</description>
 </property>
</configuration>
```

Please work with your lovely Hadoop Administrator to update the `hplsql-site.xml` file with the appropriate connection information appearing between the `<value>` and `</value>` tags. Take note that, although the full connection name above is `hplsql.conn.impala`, you can simply refer to it as `impala` within your HPL/SQL programs. For example, to issue ImpalaSQL queries within your HPL/SQL programs, code the following to connect to the database:

```
set hplsql.conn.default=impala;
```

HPL/SQL will let you know if your connection succeeded or not. For example, when HPL/SQL executes the line of code above, you should see a message similar to the following:

```
Open connection: jdbc:impala://hdpserver:21050/default (178 ms)
```

Now, the `hplsql-site.xml` file can contain many more Property Sections beside the one shown above for Impala. For example, the connection to Hive (to issue HiveQL queries) may be similar to the following:

```
<property>
  <name>hplsql.conn.hive2conn</name>
  <value>org.apache.hive.jdbc.HiveDriver;jdbc:hive2://ip-address:10000;
                                                      user;password;</value>
  <description>HiveServer2 JDBC connection</description>
</property>
```

And to MySQL starting in the `test` schema:

```
<property>
  <name>hplsql.conn.mysqlconn</name>
  <value>com.mysql.jdbc.Driver;jdbc:mysql://ip-address/
                              schema?serverTimezone=UTC;user;password;</value>
  <description>MySQL connection</description>
</property>
```

And so on for other databases. Now, regardless of the database, the connection information contained between `<value>` and `</value>` must be updated with the appropriate driver, hostname, database/schema, username and password, if necessary. Note also that if your network is running Kerberos, the connection must contain additional information such as `KrbRealm`, `KrbHostFQDN`, `KrbServiceName`, etc. For example,

```
<configuration>
 <property>
  <name>hplsql.conn.impala</name>
   <value>com.cloudera.impala.jdbc4.Driver;jdbc:impala://hdpserver:21050/prod_sc
   hema;AuthMech=1;KrbRealm=COMPANY.COM;KrbHostFQDN=hdpserver.company.com;KrbSer
   viceName=impala;</value>
  <description>Impala JDBC Connection</description>
 </property>
</configuration>
```

It may not look pretty, but it'll get the job done…just like the nachos at Taco Bell.

Chapter 26 – HPL/SQL Syntax

In this chapter, we discuss HPL/SQL syntax such as declaration blocks, `if-then-else`, `while`, `loops`, assignments, etc. Due to the volume of options available, we only show a portion of the available syntax. Please see the spiffy online HPL/SQL documentation for the full smash.

HPL/SQL Data Types

You can use the same data types from ImpalaSQL in HPL/SQL: `BOOLEAN`, `TINYINT`, `SMALLINT`, `INT`, `BIGINT`, `DECIMAL(p,s)`, `FLOAT`, `DOUBLE`, `REAL`, `CHAR(n)`, `VARCHAR(n)`, `STRING`, `DATE` and `TIMESTAMP`. Since the development team for HPL/SQL pulled in features from several different procedural languages, the following additional data types are available as well, shown below mapped to their corresponding ImpalaSQL data types:

- `BOOL` – Same as `BOOLEAN`
- `INT8` – Same as `BIGINT`
- `INT4`/`INTEGER`/`PLS_INTEGER`/`SIMPLE_INTEGER`/`BINARY_INTEGER` – Same as `INT`
- `INT2` – same as `SMALLINT`
- `BINARY_DOUBLE`/`DOUBLE PRECISION`/`SIMPLE_DOUBLE` – Same as `DOUBLE`
- `BINARY_FLOAT`/`SIMPLE_FLOAT` – Same as `FLOAT`/`REAL`
- `DATETIME` – Same as `TIMESTAMP`
- `NUMBER(p,s)`/`NUMERIC(p,s)` – Same as `DECIMAL(p,s)`
- `CHARACTER(n)` – Same as `CHAR(n)`
- `NCHAR(n)`/`NVARCHAR(n)`/`VARCHAR(n)`/`VARCHAR2(n)`/`VARCHAR(max)` – Same as `STRING`

DECLARE/BEGIN/END Block

Ignoring functions and procedures for a moment, an HPL/SQL program has the following general format:

```
...crap goes here...
DECLARE

 ...more crap goes here...

BEGIN

 ...even more crap goes here...

END;
...don't usually crap here...
```

At the top, place your desired database connection, function/procedure `INCLUDE`s, and other settings, although you can connect to and disconnect from a database at will throughout your code depending on your program's goal or general whimsy. In the `DECLARE` section, all variables used throughout the program should be defined and, if necessary, set to a value using the assignment operator `:=`. The `BEGIN` block is where your spectacular HPL/SQL code should be placed and is where the magic happens. Incredible!!

When coded like this, your HPL/SQL program will start running from the top and boogie on down, in 1970s disco stylie, to the end. For example,

```
set hplsql.conn.default=impala;
DECLARE

 iCNT int;
 sSQL string := 'SELECT COUNT(*) AS CNT FROM PROD_SCHEMA.DIM_POSTAL_CODE';
```

```
BEGIN  <-- Devoid of semicolon!

 EXECUTE(sSQL) INTO iCNT;

 DBMS_OUTPUT.PUT_LINE(iCNT);

END; <-- Undevoid of semicolon!
```

The DECLARE section allows you to define variables with or without initialization. In the code above, sSQL is initialized with some SQL code in quotes, but iCNT is not initialized. Also, note that the BEGIN keyword does not end with a semicolon, but the END keyword does. Go figure!

Now, the DECLARE/BEGIN/END block may be a great way to practice coding HPL/SQL, but in the long run you're going to need to create functions and procedures similar to how you code using your legacy database's procedural language. Although we haven't introduced basic HPL/SQL syntax yet, in the next section we let down our tresses and flail with shameless abandon through the fields of functions and procedures.

Hairless Functions and Comb-Over Procedures

The neutered DECLARE/BEGIN/END block described previously will only get you so far and, let's face it, lacks a certain *je ne sais panache*. In this section, we discuss how to create functions and procedures in HPL/SQL as well as how to call them.

To create a **function** in HPL/SQL, the basic syntax is as follows:

```
CREATE OR REPLACE FUNCTION function_name(IN pARG1 datatype1,...)
                                                      RETURNS datatype AS

...variable declarations...

BEGIN

 ...your code here...

 RETURN variable;

END;
```

Take note of the IN keyword. This indicates the direction of the parameter value when the function is called: **IN**PUT. Since HPL/SQL doesn't allow for the OUT keyword (**OUT**PUT) with a function, only IN should be provided.

The nice folks at *HPL/SQL Global Planetary World Domination Headquarters* also allow you to place the IN keyword **after** the parameter name, but **before** its associated data type, if you prefer: *pARG1* **IN** *datatype1*. HPL/SQL syntax also allows you to use the RETURN keyword as a synonym for the RETURNS keyword depending on your plurality. You can also use the keyword IS instead of AS, but that's just going too far now!

To call a function named, say, **MYFTN()**, simply use it as you would a normal function passing in the appropriate number of parameters, here shown with the useful built-in PRINT Statement:

```
PRINT MYFTN('TESTING 1, 2, 3!');
```

To create a **procedure** in HPL/SQL, the basic syntax is as follows:

```
CREATE OR REPLACE PROCEDURE procedure_name(direction1 pARG1 datatype1,...) AS

...variable declarations...

BEGIN

 ...your code here...

END;
```

Take note of the term `direction1` in the syntax above. This indicates the direction of the parameter value when the procedure is called. HPL/SQL procedures allow for the `IN`, `OUT`, `INOUT` shake-it-all-about keywords indicating the following:

- ☐ `IN` – the parameter value is being provided from the call into the procedure
- ☐ `OUT` – the parameter value is being pushed from within the procedure itself and can be captured in the code following the call to the procedure. This is nice because procedures, unlike functions, don't allow for a return value.
- ☐ `INOUT` – a combination of the two, `IN` and `OUT`; that is, you can provide a parameter value going into the procedure, but the procedure itself can alter that value which you can then capture after the procedure has completed.

Unlike functions, to call a procedure, say, `MYPROC`, use the `CALL` Statement:

```
CALL MYPROC('TESTING 1, 2, 3!');
```

Note that, while testing a function or procedure, you can embed the call to it within the same `.hplsql` file. But, don't leave the call in the file when you're done testing. Instead, create a completely separate `.hplsql` file to act as a *stub*, `INCLUDE` the function or procedure at the top of the *stub* using the `INCLUDE` Statement, and then call the function or procedure from the *stub*. Why? If you leave the call within your function's or procedure's `.hplsql` file, when you `INCLUDE` the `.hplsql` file later on as part of a larger system of functions and procedures, **the call will immediately execute**, which is something you probably don't want to happen. This is especially important if you're planning to execute your HPL/SQL programs from, say, your fab department's magical dynamical website. Below is an example *stub* used to call the procedure `MYPROC` (located in the `myproc.hplsql` file):

```
set hplsql.conn.default=impala;
include /directory/location/hplsql/files/myproc.hplsql

declare
begin

 call myproc('TESTING 1, 2, 3!');

end;
```

Stub…that's a funny word!

Conditional Execution

You can use the `IF-THEN-ELSE` Statement to conditionally execute code based on a condition. The basic `IF` Statement looks like this:

```
IF boolean-expression THEN

 ...statements...

END IF;
```

You can include additional conditions by adding the ELSEIF Condition one or more times:

```
IF boolean-expression THEN
  ...statements...

ELSEIF boolean-expression-1 THEN

 ...statements...

ELSEIF boolean-expression-2 THEN

 ...statements...

END IF;
```

Finally, you can add one massively incredible ELSE Clause to supplant them all:

```
IF boolean-expression THEN

 ...statements...

ELSEIF boolean-expression-1 THEN

 ...statements...

ELSEIF boolean-expression-2 THEN

 ...statements...

ELSE

 ...statements...

END IF;
```

Note that HPL/SQL syntax allows you to use the keyword ELSIF as a synonym for ELSEIF dpnding on your mdically untratd hatrd of the lttr E.

Looping Constructs

HPL/SQL has the traditional looping constructs such as FOR, WHILE, and LOOP.

The FOR Statement allows you to repeat statements a specified number of times. The general syntax is as follows:

```
FOR index-variable IN starting-value..ending-value LOOP

 ...statements...

END LOOP;
```

Note that *index-variable* is implicitly declared as an integer and can be used within the containing statements. By default, the loop will increment the *index-variable* by one from *starting-value* to *ending-value*, but you can change this by providing the STEP keyword followed by a step value:

```
FOR index_variable IN starting-value..ending-value STEP step-value LOOP

 ...statements...

END LOOP;
```

Also, if you're a rebel and prefer the loop to decrement instead of increment, you can provide the `REVERSE` keyword:

```
FOR index_variable IN REVERSE starting-value..ending-value LOOP

 ...statements...

END LOOP;
```

If you need to loop while a condition is true, you can use a `WHILE` Statement. The basic syntax is as follows:

```
WHILE boolean-expression LOOP

 ...statements...

END LOOP;
```

As long as the *`boolean-expression`* is true, the loop will continue to…uh…loop. Note that the HPL/SQL syntax allows for variations such as `DO` and `BEGIN` as synonyms for the keyword `LOOP` as well as `END WHILE` as a synonym for `END LOOP`. So, pick your poison, kids!

If you're more old school and like to take control of any situation, you can just use the `LOOP` Statement to roll your own doobie-smack:

```
LOOP

 ...statements...

END LOOP;
```

Now, the loop shown above will trudge on indefinitely (like the more recent *Star Wars* movies), but you can end the loop prematurely by using one of the following nifty keywords:

- `EXIT WHEN` *`boolean-expression`* – The loop will end when the *`boolean-expression`* is true.
- `LEAVE` – The loop will end when this keyword is reached. You can surround this keyword by an `IF` Statement for more control.
- `BREAK` – The ***innermost*** loop will end when this keyword is reached. You can surround this keyword by an `IF` Statement for more control. Note that if there's only one loop, `BREAK` behaves similar to `LEAVE`.

Arithmetic Operators

HPL/SQL has the traditional arithmetic operators such as `+` (addition), `-` (subtraction), `*` (multiplication) and `/` (division). Be careful when performing division! If the two operators are integers, then your result will be an integer with any fractional part flamethrowered off. Note that HPL/SQL doesn't provide for the modulus (`%`) operator.

Assignment Operators

HPL/SQL has the traditional assignment operators such as `=` and `:=`, as indicated earlier. You can also use the `SET` keyword to assign values to one or more variables:

```
SET variable-name-1 = value-1;
```

...and is equivalent to...

```
variable-name-1 := value-1;
```

You can assign to more than one variable at a time, like this:

```
SET variable-name-1 = value-1,   COMMA!!
    variable-name-2 = value-2,   COMMA!!
    ...,   COMMA!!
    variable-name-n = value-n;
```

Alternatively, you can use the following bangin' syntax to assign values to variables in one *swell foop*:

```
SET (variable-name-1,variable-name-2,...) = (value-1,value-2,...);
```

Yummy!

Comparison Operators

HPL/SQL has the traditional comparison operators such as = (is equal to), != (is not equal to), <> (is not equal to), < (is less than), > (is greater than), <= (is less than or equal to) and >= (is greater than or equal to).

Logical Operators

HPL/SQL has the traditional logical operators such as `AND` (Logical AND), `OR` (Logical OR) and `NOT` (negation). Note that the `NOT` Logical Operator must surround its victim by parentheses. In any case, the judicious use of parentheses will make your life rosier and keep you smelling fresh all day. For example:

```
IF (iCNT1 < iCNT2) AND (iCNT1 < iCNT2) THEN
 PRINT 'HOORAY!';
END IF;

IF (iCNT1 < iCNT2) OR (iCNT1 < iCNT2) THEN
 PRINT 'HOORAY!';
END IF;

IF NOT(iCNT1 > iCNT2) THEN
 PRINT 'HOORAY!';
END IF;
```

Ahem! I'm Talkin' Here! (Making Comments)

Inline comments are indicated by two dashes:

```
--This is an inline comment.
```

If you're feeling lateral, an inline comment can be placed at the end of a line of code as well:

```
iCNT INT := 0; --This is an inline comment.
```

Multiline comments are indicated by using **/*** and ***/**:

```
/* The following code will not run because it`s part of a multiline comment:
iCNT1 INT := 1;
iCNT2 INT := 2;
iCNT3 INT := 3;
*/
```

Voyage of the Damned (Dates & Times – HPL/SQL Edition)

HPL/SQL allows you to operate natively on those damnable dates and times including the availability of the `INTERVAL` keyword similar to ImpalaSQL. Natively, HPL/SQL has the following `DATE` and `TIMESTAMP` literals:

```
DATE 'YYYY-MM-DD'
TIMESTAMP 'YYYY-MM-DD HH:MI:SS.sss'
```

The first can be used where `DATE`s are expected; the second, where `TIMESTAMP`s are expected. But, I expect you expected that already.

HPL/SQL allows for several date/time-related data types, as we've seen earlier, such as `DATE`, `TIMESTAMP` and `DATETIME`. These data types, of course, can be used to initialize HPL/SQL variables.

You can easily kick dates and timestamps around by using the `INTERVAL` keyword:

```
INTERVAL shift-amount DAYS
INTERVAL shift-amount MICROSECONDS
```

Note that HPL/SQL allows for the keyword `DAY` as a synonym for `DAYS` and `MICROSECOND` as a synonym for `MICROSECONDS`. For example, let's shift the date March 21, 1962 forward by one day:

```
PRINT DATE '1962-03-21' + INTERVAL 1 DAY;
1962-03-22
```

Working with Strings

There are several operators and functions you can use to concatenate strings. The addition operator (`+`) can be used to concatenate two strings, like this:

```
sSTR1 := 'HELLO ';
sSTR2 := 'WORLD!';
sSTR := sSTR1 + sSTR2;
```

Two vertical bars (`||`) perform the same function:

```
sSTR := sSTR1 || sSTR2;
```

HPL/SQL allows you to create a sweet multiline string, like this:

```
sSQL := "
         SELECT *
          FROM PROD_SCHEMA.DIM_POSTAL_CODE
          WHERE STATE_CODE='PA'
          ORDER BY POSTAL_CODE
        ";
```

The syntax above allows you to use the contatenation operators `+` and `||` as well:

```
sSQL := "
         SELECT *
          FROM PROD_SCHEMA.DIM_POSTAL_CODE
          WHERE STATE_CODE='" || sSTATE_CODE || "'
          ORDER BY POSTAL_CODE
        ";
```

Finally, the HPL/SQL CONCAT() function can concatenate two or more strings:

```
sSTR := CONCAT(sSTR1,sSTR2);
```

We talk more about HPL/SQL functions later.

Later is Now, Baby! HPL/SQL Functions

Unlike other procedural languages such as Oracle PL/SQL and Microsoft SQL Server T-SQL, HPL/SQL's list of available functions is a bit of a tiddler. With that said, be aware that you can access the full list of ImpalaSQL functions via the EXECUTE Statement when you're connected to the database. We discuss this later on. Below is a list of the available HPL/SQL functions:

HPL/SQL FUNCTIONS		
Function	**Return Type**	**Description**
CAST(*expression* AS *datatype*)	*datatype*	Returns ***expression*** cast to the desired ***datatype***.
CHAR(*numeric-expression*)	STRING	Converts ***numeric-expression*** to a STRING.
COALESCE(*expr-1*,*expr-2*,...,*expr-n*) **NVL(*expr-1*,*expr-2*,...,*expr-n*)**	***expr-i***	Returns the first non-NULL expression, ***expr-i***, as ***expr-i***'s data type.
CONCAT(*expr-1*,*expr-2*,...,*expr-n*)	STRING	Returns a concatenation of all ***expr-i***'s.
CURRENT_DATE **CURRENT DATE**	DATE	Returns the current date. Take note that no function parentheses are required.
CURRENT_TIMESTAMP **CURRENT TIMESTAMP**	TIMESTAMP	Returns the current date/time. Take note that no function parentheses are required, but if you require fractions of seconds, you can append (#) where # is from 0 to 3 digits of fractional seconds.
CURRENT_USER **CURRENT USER** **USER**	STRING	Returns the username of the poor working stiff who's executing hplsql at the time. Take note that no function parentheses are required.
DATE('YYYY-MM-DD') **DATE(*timestamp*)**	DATE	Converts a text string in the format YYYY-MM-DD into a DATE. Can also be used to convert a TIMESTAMP data type into a DATE data type.
DECODE(*expression*, ***when-1*,*then-1*,** ***when-2*,*then-2*,** **...,** ***when-n*,*then-n*,** ***then-n+1*)**	***then-i***	Returns the value of ***then-i*** if the corresponding ***when-i*** matches ***expression***. If not, so sad, but ***then-n+1*** is returned instead.
DBMS_OUTPUT.PUT_LINE(*string*) **PRINT(*string*)** **PRINT *string***	N/A	Prints ***string*** to the STDOUT.
FROM_UNIXTIME(*seconds-off-epoch*, ***format-string*)**	STRING	Returns a STRING as specified in ***format-string*** of the number of seconds off the Unix Epoch (1970-01-01 00:00:00). If you leave off ***format-string***, the default yyyy-MM-dd HH:mm:ss format is used.
INSTR(*string*, ***search-string*,** ***starting-position*,** ***occurrence-number*)**	INT	Returns the starting position of ***search-string*** within the ***string***. If ***search-string*** is not found, 0 is returned. By default, ***string*** is searched starting from the first position, but this can be altered by specifying ***starting-position***. If ***search-string*** occurs more than once, you can return a specific occurrence by specifying ***occurrence-number***. Both ***starting-position*** and ***occurrence-number*** can be left off the function call and their default values of 1 will be used.
LOWER(*expression*)	STRING	Returns ***expression*** lowercased.
LEN(*string*) **LENGTH(*string*)**	INT	Returns the length of ***string*** with the following caveats: ☐ LEN **excludes** leading and trailing blanks ☐ LENGTH **includes** leading and trailing blanks
NOW()	TIMESTAMP	Returns the current date and time with fractional seconds.
NVL2(*expr-1*,*expr-2*,*expr-3*)	***expr-2*** ***expr-3***	If ***expr-1*** is not-NULL, ***expr-2*** is returned. If ***expr-1*** is NULL, ***expr-3*** is returned.
REPLACE(*string*, ***search-term*,**	STRING	Returns ***string*** with the ***search-term*** replaced by ***replace-term***.

***replace-term*)**		
SUBSTR(*string*, *starting-position*, *substring-length*) **SUBSTRING(*string*, *starting-position*, *substring-length*)**	`STRING`	Returns a substring of length ***substring-length*** of ***string*** starting at the desired ***starting-position***. If ***substring-length*** is left off, a substring from ***starting-position*** to the end of the string is returned.
SYSDATE	`TIMESTAMP`	Returns the current date/time.
TIMESTAMP_ISO(*expression*)	`TIMESTAMP`	Returns a `TIMESTAMP` from ***expression***. If ***expression*** is a `STRING`, it must be formatted as either `YYYY-MM-DD` or `YYYY-MM-DD HH24:MI:SS.FF`. If ***expression*** is a `DATE`, then ***expression*** is converted to a `TIMESTAMP`.
TO_CHAR(*expression*)	`STRING`	Converts ***expression*** to a `STRING`.
TO_TIMESTAMP(*string*, *format-string*)	`TIMESTAMP`	Converts ***string*** in the format specified by ***format-string*** into a `TIMESTAMP`. Unlike the ImpalaSQL-related formats, this ***format-string*** can only take on the following paltry number of elements: ☐ `YYYY` – Four-digit year ☐ `MM` – One or two-digit month (`1` to `12` or `01` to `12`) ☐ `DD` – Day of month (`1` to `31` or `01` to `31`) ☐ `HH24` – Hour of the day as a `24`-hour clock (`0` to `23` or `00` to `23`) ☐ `MI` – Minutes of the hour (`0` to `59` or `00` to `59`) ☐ `SS` – Seconds of the minute (`0` to `59` or `00` to `59`)
TRIM(*string*)	`STRING`	Returns ***string*** with both leading and trailing blanks hacked off.
UNIX_TIMESTAMP()	`INT`	Returns the current date/time as the number of seconds since Unix Epoch.
UPPER(*expression*)	`STRING`	Returns ***expression*** uppercased.

HPL/SQL Example #1 – The First of the Examples

Recall from college calculus class that the following is true:

$$\int_0^{10} x^2 dx = \left.\frac{x^3}{3}\right|_0^{10} = 333.3\overline{3}$$

[We'll wait for those of you hyperventilating right now…you 'kay?…good!…moving on…]

Let's try to approximate this definite integral using HPL/SQL by dividing up the area under x^2 into a specified number of rectangles, n, each with a width given by $h = (x_1 - x_0)/n$, where $x_1 = 10$ and $x_0 = 0$. Here's a first cut of the HPL/SQL code `integrate_x2.hplsql`:

```
DECLARE

 X0 DOUBLE := 0;
 X1 DOUBLE := 10;
 H DOUBLE;
 N INT;
 TOT_AREA DOUBLE := 0;
 X0_INCR DOUBLE := X0;

BEGIN

 --DEFINE N, THE NUMBER OF RECTANGLES.
 N := 1000000;

 --COMPUTE H BASED ON X0, X1 AND N. THIS IS THE INCREMENT AMOUNT
 -- TO BE APPLIED TO X0_INCR AS WELL AS THE WIDTH OF EACH RECTANGLE.
 H := (X1 - X0)/N;

 --LOOP AROUND EACH OF THE N RECTANGLES SUMMING UP THE AREA
 -- BASED ON THE RECTANGULAR RULE.
 FOR i IN 1..N LOOP
```

```
  --COMPUTE THIS ITERATION`S TOTAL AREA BY MULTIPLYING THE
  -- WIDTH (H) OF EACH RECTANGLE BY THE HEIGHT.
  TOT_AREA := TOT_AREA + H*(X0_INCR*X0_INCR);

  --SLIDE XO_INCR TO THE RIGHT BY H. PROBABLY SHOULD ADD AN EXTRA HALF
  --TO HIT THE CENTER OF THE RECTANGLE, BUT I DON'T HAVE THE STRENGTH.
  X0_INCR := X0_INCR + H;

 END LOOP;
 PRINT TOT_AREA;

END;
```

Running this at the Linux command line…

```
hplsql -f integrate_x2.hplsql
```

…produces an estimated area of…

```
333.3328333242384
```

That's not bad! Unfortunately, we had to hardcode some values, so let's try to pass in values from the `hplsql` command line using `--define` to pass in n, x_0 and x_1. Here's how we'll call the updated program:

```
[smithbob@lnxserver ~]$ hplsql -f integrate_x2.hplsql --define pX0=0
                                                      --define pX1=10
                                                      --define pN=1000000
```

The code for `X0`, `X1` and `N` above has been updated to the following:

```
X0 DOUBLE := CAST(pX0 AS DOUBLE);
X1 DOUBLE := CAST(pX1 AS DOUBLE);
N INT := CAST(pN AS INT);
```

Note that capturing the arguments passed into an HPL/SQL program is simple: just reference the define variable names. Joyous!!

And, the following lines have been removed since we're capturing `N` above now:

```
--DEFINE N, THE NUMBER OF RECTANGLES.
N := 1000000;
```

Naturally, you should code in some ***hiiii-yaaaa!*** to ensure that `pX0`, `pX1` and `pN` exist, contain valid values, etc.

HPL/SQL Example #2 – The Second of the Examples

That first example is truly well and good, but we probably should use our grownup coding skills and stick in some functions and procedures. In this example, we create a function which performs x^2 as well as a procedure to do the dirty work of computing the area. Here's what the code `integrate_x2.hplsql` looks like now:

```
CREATE OR REPLACE FUNCTION MYFTN(pX IN DOUBLE) RETURN DOUBLE AS

BEGIN

 -- COMPUTE x^2 as x*x
 RETURN pX * pX;
```

```
END;
CREATE OR REPLACE PROCEDURE COMPUTE_AREA(pX0 IN DOUBLE,
                                         pX1 IN DOUBLE,
                                         pN IN DOUBLE) AS

 -- DEFINE VARIABLES HERE!! THIS IS EFFECTIVELY THE DECLARE SECTION!
 X0 DOUBLE;
 X1 DOUBLE;
 H DOUBLE;
 N INT;
 TOT_AREA DOUBLE;
 X0_INCR DOUBLE;

BEGIN

 -- ASSIGN TO THE VARIABLES HERE!!
 X0 := CAST(pX0 AS DOUBLE);
 X1 := CAST(pX1 AS DOUBLE);
 N := CAST(pN AS INT);
 TOT_AREA := 0;
 X0_INCR := X0;

 --COMPUTE H BASED ON X0, X1 AND N. THIS IS THE INCREMENT AMOUNT
 -- TO BE APPLIED TO X0_INCR AS WELL AS THE WIDTH.
 H := (X1 - X0)/N;

 --LOOP AROUND EACH OF THE n RECTANGLES SUMMING UP THE AREA
 -- BASED ON THE RECTANGULAR RULE.
 FOR i IN 1..N LOOP

  --COMPUTE THIS ITERATION`S TOTAL AREA BY MULTIPLYING THE
  -- WIDTH (H) OF EACH RECTANGLE BY THE HEIGHT.
  TOT_AREA := TOT_AREA + H*MYFTN(X0_INCR);

  --SLIDE XO_INCR TO THE RIGHT BY H.
  X0_INCR := X0_INCR + H;

 END LOOP;

 PRINT TOT_AREA;

END;

--CALL THE PROCEDURE
CALL COMPUTE_AREA(0,10,1000000);
```

In this example, we created a function called `MYFTN()` which takes a parameter as a `DOUBLE` and returns the square of it as a double. We also created a procedure called `COMPUTE_AREA()` which is called using the `CALL` Statement passing in our desired values. Take note that for both the function `MYFTN()` and the procedure `COMPUTE_AREA()` we're using the `IN` parameter direction to provide values into both. The resulting area in the variable `TOT_AREA` is printed out using the `PRINT()` function. The result is the same as the previous example.

Now, let's modify the procedure so that the total area can be captured for use later in the program. To do this, we add an **`OUT`** parameter to the procedure `COMPUTE_AREA()`:

```
CREATE OR REPLACE PROCEDURE COMPUTE_AREA(pX0 IN DOUBLE,
                                         pX1 IN DOUBLE,
                                         pN IN DOUBLE,
                                         pOUT_TOTAREA OUT DOUBLE) AS
```

We removed the `PRINT TOT_AREA;` code and replaced it with the following code which updates the `OUT` parameter `pOUT_TOTAREA` with the final estimated area `TOT_AREA` instead:

```
pOUT_TOTAREA := TOT_AREA;
```

Now, the call to the procedure needs some housekeeping, shown in bold font:

```
--CALL THE PROCEDURE
DECLARE dTOTAREA DOUBLE;
CALL COMPUTE_AREA(0,10,1000000,dTOTAREA);
PRINT dTOTAREA;
```

We first declare a variable, `dTOTAREA`, which will receive the final estimated area from the `OUT` parameter. And the resulting value, no surprise, is `333.3328333242384`.

Now, if you still want to pass in values from the `hplsql` command line using `--define`, we can create a *stub* which will then call the procedure. So, remove the following lines from `integrate_x2.hplsql`:

```
--CALL THE PROCEDURE
DECLARE dTOTAREA DOUBLE;
CALL COMPUTE_AREA(0,10,1000000,dTOTAREA);
PRINT dTOTAREA;
```

Next, create a *stub* called `stubby.hplsql` containing the following code:

```
INCLUDE /directory/location/hplsql/files/integrate_x2.hplsql
DECLARE

 X0 DOUBLE := CAST(pX0 AS DOUBLE);
 X1 DOUBLE := CAST(pX1 AS DOUBLE);
 N INT := CAST(pN AS INT);
 dTOTAREA DOUBLE;

BEGIN

 CALL COMPUTE_AREA(X0,X1,N,dTOTAREA);
 PRINT dTOTAREA;

END;
```

Note that the function `MYFTN()` and the procedure `COMPUTE_AREA()`, both located in the file `integrate_x2.hplsql`, are included into the *stub* using the `INCLUDE` Statement. At this point, both are available to use in your code. Note that `stubby.hplsql` is executed in the usual manner from the Linux command line:

```
hplsql -f stubby.hplsql --define pX0=0 --define pX1=10 --define pN=1000000
```

And the passed in parameters are used in the code just as before.

A Chat about Packages

In Oracle's procedural language PL/SQL, functions and procedures can be combined into one giant programmy goodness known as a *package*. HPL/SQL has similar functionality! A package is actually made up of a *package specification*, which defines all of the variables, functions and procedures that make up the package, but doesn't contain any code. The corresponding code is actually placed in the *package body* instead. Both the *package specification* and the *package body* are created using different HPL/SQL statements. The general syntax for a *package specification* is as follows:

```
CREATE OR REPLACE PACKAGE package_name AS

...variable declarations...

...function declarations...

...procedure declarations...

END;
```

Once the *package specification* has been defined, you can write the code for your package within a *package body*. The general syntax for a *package body* is as follows:

```
CREATE OR REPLACE PACKAGE BODY package_name AS

...variable declarations...

...function code...

...procedure code...

END;
```

Both the *package specification* and the *package body* can be placed in the same file. Fantastic!!

In the next example, we update our integration example by creating a package called `PKG_COMPUTE_AREA`.

HPL/SQL Example #3 – The Third of the Examples

Let's create a *package specification* and *package body* to contain our integration code:

```
/* PACKAGE SPECIFICATION */
CREATE OR REPLACE PACKAGE PKG_COMPUTE_AREA AS

 /* VARIABLE DECLARATION */
 X0 INT; --INITIAL X-VALUE
 X1 DOUBLE; --ENDING X-VALUE
 H DOUBLE; --RECTANGLE WIDTH
 N INT; --NUMBER OF RECTANGLES
 TOT_AREA DOUBLE; --TOTAL AREA

 /* FUNCTION DECLARATION(S) */
 --FUNCTION TO RETURN THE SQUARE OF THE ARGUMENT
 FUNCTION MYFTN(pX IN DOUBLE) RETURN DOUBLE;

 --FUNCTION TO RETURN THE COMPUTED AREA
 FUNCTION GET_AREA() RETURN DOUBLE;

 /* PROCEDURE DECLARATION(S) */
 --PROCEDURE TO COMPUTE THE AREA UNDER THE CURVE
 PROCEDURE COMPUTE_AREA(pX0 IN DOUBLE,pX1 IN DOUBLE,pN IN DOUBLE);

END;


/* PACKAGE BODY */
CREATE OR REPLACE PACKAGE BODY PKG_COMPUTE_AREA AS

 /* VARIABLE DECLARATION(S) */
```

```
    X0_INCR DOUBLE;
    /* FUNCTION(S) */
    --FUNCTION TO RETURN THE SQUARE OF THE ARGUMENT
    CREATE OR REPLACE FUNCTION MYFTN(pX IN DOUBLE) RETURN DOUBLE AS

    BEGIN

     RETURN pX * pX;

    END;

    --FUNCTION TO RETURN THE COMPUTED AREA
    CREATE OR REPLACE FUNCTION GET_AREA() RETURN DOUBLE AS

    BEGIN

     RETURN TOT_AREA;

    END;

    /* PROCEDURE(S) */
    CREATE OR REPLACE PROCEDURE COMPUTE_AREA(pX0 IN DOUBLE,
                                             pX1 IN DOUBLE,
                                             pN IN DOUBLE) AS

    BEGIN

     -- ASSIGN TO THE VARIABLES HERE.
     X0 := CAST(pX0 AS DOUBLE);
     X1 := CAST(pX1 AS DOUBLE);
     N := CAST(pN AS INT);
     TOT_AREA := 0;
     X0_INCR := X0;

     --COMPUTE H BASED ON X0, X1 AND N. THIS IS THE INCREMENT AMOUNT
     -- TO BE APPLIED TO X0_INCR AS WELL AS THE WIDTH.
     H := (X1 - X0)/N;

     --LOOP AROUND EACH OF THE n RECTANGLES SUMMING UP THE AREA
     -- BASED ON THE RECTANGULAR RULE.
     FOR i IN 1..N LOOP

      --COMPUTE THIS ITERATION`S TOTAL AREA BY MULTIPLYING THE
      -- WIDTH (H) OF EACH RECTANGLE BY THE HEIGHT.
      TOT_AREA := TOT_AREA + H*MYFTN(X0_INCR);

      --SLIDE XO_INCR TO THE RIGHT BY H.
      X0_INCR := X0_INCR + H;

     END LOOP;

    END;

   END;
```

In the code above, the *package specification* only **defines** the variables, functions and procedures. Also, note that the syntax `CREATE OR REPLACE` is left off both the function and procedure in the *package specification*. The code for the functions and procedures is located in the *package body*. Note that for the procedure `COMPUTE_AREA()`, the `OUT` variable was removed and a new function, `GET_AREA()`, was created to retrieve the computed area. Below is the *stub* used to run the package to compute the area:

```
INCLUDE /directory/location/hplsql/files/integrate_x2.hplsql
DECLARE

 dTOTAREA double;

BEGIN

 --COMPUTE THE AREA UNDER X^2 FROM 0 TO 10 USING 1,000,000 RECTANGLES.
 PKG_COMPUTE_AREA.COMPUTE_AREA(0,10,1000000);

 --RETRIEVE THE COMPUTED AREA USING THE GET_AREA() FUNCTION.
 dTOTAREA := PKG_COMPUTE_AREA.GET_AREA();

 PRINT dTOTAREA;

END;
```

Note that you don't need to use the `CALL` Statement to execute the procedure `COMPUTE_AREA()` in a package.

In the call to the `COMPUTE_AREA()`, we're still passing in the same three values as parameters. You could include additional functions to the *package specification* as well as *package body* as *getters* and *setters* for the values for n, x_0 and x_1 instead. With that said, you can retrieve these values, without creating *getters*, like this:

```
PRINT PKG_COMPUTE_AREA.X0;
PRINT PKG_COMPUTE_AREA.X1;
PRINT PKG_COMPUTE_AREA.N;
PRINT PKG_COMPUTE_AREA.H;
PRINT PKG_COMPUTE_AREA.X0_INCR;
```

Chapter 27 – HPL/SQL and Chatting with a Database

So far, we've been playing around with HPL/SQL syntax to compute an approximate area under the curve of x^2. Although we've learned how to create functions, procedures, packages as well as some basic HPL/SQL syntax, in this chapter we focus on how to interact with a database from an HPL/SQL program.

Now, there are several methods you can employ here.

The first method is to create a string containing your SQL code and then submit that code directly to the database using the `EXECUTE` Statement. Many programmers prefer this method because you're assured that no manipulation of your SQL code occurs by HPL/SQL. In other words: *in a corn, out a corn*.

The second method is to issue SQL statements directly within your HPL/SQL code. Statements such as `CREATE TABLE`, `DROP TABLE`, etc. are available to use within your programs. These statements are passed along to the database to be executed.

The third method to interact with a database is with *cursors*, a mechanism which allows you to submit a SQL query and retrieve the results by processing them one row at a time.

We discuss all three methods below, but first this message…

Switching Schemas or *Where Am I Now?*

When you connect to a database using the following syntax…

```
set hplsql.conn.default=impala;
```

…you're implicitly connecting to a specific schema based on the information contained within the `hplsql-site.xml` file. For the line of code shown above, the corresponding XML from `hplsql-site.xml` is shown below (your own XML will be slightly different):

```
<configuration>
 <property>
  <name>hplsql.conn.impala</name>
  <value>com.cloudera.impala.jdbc4.Driver;jdbc:impala://hdpserver:21050/
                                        default;smithbob;PaSsWoRd123;</value>
  <description>Impala JDBC Connection</description>
 </property>
</configuration>
```

Take note that the schema connected to automatically is the **`default`** database, shown above in bold font. If you'd prefer to connect automatically to, say, the **`prod_schema`** schema instead, you or your highly-favorable Hadoop Administrator can code the XML above like this:

```
<configuration>
 <property>
  <name>hplsql.conn.impala</name>
  <value>com.cloudera.impala.jdbc4.Driver;jdbc:impala://hdpserver:21050/
                                    prod_schema;smithbob;PaSsWoRd123;</value>
  <description>Impala JDBC Connection</description>
 </property>
</configuration>
```

Now, when you issue the line of code…

```
set hplsql.conn.default=impala;
```

…you'll connect automatically to `prod_schema`. Hoo-haa!!

But, you don't necessarily have to do that since HPL/SQL allows you to change the current schema at will by tapping on the shoulder of the `USE` Statement. For example, let's connect to the `prod_schema` database:

```
use prod_schema;
```

Be aware that the connection is still to Impala, but the current schema is now `prod_schema`. This is similar to how you change schemas when working in `impala-shell` and is similar to how you change schemas in other databases, such as MySQL, Microsoft SQL Server, etc.

Another way to do the same thing is to use the `EXECUTE` Statement, which we describe in more detail in the next section, to pass SQL code directly to the database:

```
sSQL := "use prod_schema";
EXECUTE sSQL;
```

Beautious!!

Off With His Head! (The `EXECUTE` Statement)

You can pass SQL code directly to the database by using the `EXECUTE` Statement which takes the following general format:

```
EXECUTE ddl-sql-string;
```

Note that HPL/SQL allows you to use `EXEC` and `EXECUTE IMMEDIATE` as synonyms for `EXECUTE`. For example, let's create a backup table of the `DIM_POSTAL_CODE` table:

```
sSQL := "
         CREATE TABLE DIM_POSTAL_CODE_BACKUP STORED AS PARQUET AS
          SELECT *
           FROM DIM_POSTAL_CODE
        ";
EXECUTE sSQL;
```

Naturally, you can pass any SQL code directly to the database like this. Another reason to run SQL code using the `EXECUTE` Statement is to allow for dynamic SQL code which changes based on, say, a selection from a website. In the example below, our dynamic SQL code, placed in a variable called `sSQL_TMPL`, is used to create the final SQL code which is then submitted directly to the database:

```
sSQL_TMPL := "
              CREATE TABLE DIM_POSTAL_CODE_BACKUP_stcd STORED AS PARQUET AS
               SELECT *
                FROM DIM_POSTAL_CODE
                WHERE STATE_CODE='stcd'
             ";

--The selected state code is coming in from our fab dept website!
sSQL := REPLACE(sSQL_TMPL,"stcd","PA");
EXECUTE sSQL;
```

Naturally, you can code using the string concatenation operators `+` or `||` instead of using the `REPLACE()` function.

Another nice feature is the ability to receive ***exactly one row back*** from a SQL query. In other words, it's not always about `CREATE TABLE`, `DROP TABLE`, etc., but `SELECT` as well! The general syntax for this form of `EXECUTE` is as follows:

```
EXECUTE dml-sql-string INTO hplsql-var1, hplsql-var2,...;
```

Recall I mentioned that the list of HPL/SQL functions, as compared to the list of ImpalaSQL functions, is a bit of a tiddler; in other words, there ain't many functions. But, you have access to ImpalaSQL's entire palette of functions by using the form of the `EXECUTE` Statement shown above. Again, be aware that if the SQL query in *`dml-sql-string`* returns more than one row, only the first row's data will be inserted into the variables *`hplsql-var1`*, *`hplsql-var2`*, etc.

For example, let's compute the total number of rows and total number of distinct `STATE_CODE`s in the table `DIM_POSTAL_CODE`:

```
set hplsql.conn.default=impala;
DECLARE

 iROWCNT int;
 iSTCDCNT int;
 sSQL string;

BEGIN

 sSQL := "use prod_schema";
 EXECUTE sSQL;

 sSQL := "
          SELECT COUNT(*) AS ROWCNT,
                 COUNT(DISTINCT STATE_CODE) AS DISTSTCD
           FROM DIM_POSTAL_CODE
         ";
 EXECUTE sSQL INTO iROWCNT,iSTCDCNT;

 PRINT "DIM_POSTAL_CODE has " || TO_CHAR(iROWCNT) || " rows.";
 PRINT "DIM_POSTAL_CODE has " || TO_CHAR(iSTCDCNT) || " distinct state codes.";

END;
```

Note that the first column in the query (`ROWCNT`) maps to the first HPL/SQL variable (`iROWCNT`), and so on. And running this code, stored in the file `counts.hplsql`, at the Linux command line yields the following output:

```
[smithbob@lnxserver ~]$ hplsql -f counts.hplsql
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
Open connection: jdbc:impala://hdpserver:21050/default (140 ms)
Starting SQL statement
SQL statement executed successfully (76 ms)
Starting SQL statement
SQL statement executed successfully (245 ms)
DIM_POSTAL_CODE has 43689 rows.
DIM_POSTAL_CODE has 61 distinct state codes.
```

Finally, if you pass a `SELECT` Statement to the database through the `EXECUTE` Statement, the results will be displayed:

```
set hplsql.conn.default=impala;
DECLARE

 sSQL string;
```

```
BEGIN

 sSQL := "use prod_schema";
 EXECUTE sSQL;

 sSQL := "
          SELECT *
           FROM DIM_POSTAL_CODE
           WHERE STATE_CODE='NJ'
           ORDER BY POSTAL_CODE
           LIMIT 10
         ";
 EXECUTE sSQL;

END;

SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
Open connection: jdbc:impala://hdpserver:21050/default (141 ms)
Starting SQL statement
SQL statement executed successfully (76 ms)
Starting SQL statement
SQL statement executed successfully (256 ms)
07001   AVENEL            NJ    40.57899    -74.27987
07002   BAYONNE           NJ    40.66655    -74.11768
07003   BLOOMFIELD        NJ    40.80300    -74.18895
07004   FAIRFIELD         NJ    40.87904    -74.29378
07005   BOONTON           NJ    40.91279    -74.41516
07006   CALDWELL          NJ    40.84899    -74.27917
07007   CALDWELL          NJ    40.79185    -74.24524
07008   CARTERET          NJ    40.58250    -74.22997
07009   CEDAR GROVE       NJ    40.85585    -74.22898
07010   CLIFFSIDE PARK    NJ    40.82154    -73.98949
```

Amazing!!

HPL/SQL Example #4 – The Fourth of the Examples

A useful table to have available in any database is a calendar with useful date-related information such as the year, month, day, quarter, etc. as well as a variety of formats such as `YYYYMMDD` and `YYYYMM`. In example below, we create the table `DIM_CALENDAR` and populate it with data. Here's a preliminary definition for our calendar table:

DIM_CALENDAR		
COLUMN NAME	**DATA TYPE**	**DESCRIPTION**
DATE_ID	DATE	DATE_ID serves as the primary key of the table.
DAY	TINYINT	DATE_ID`s day value (1-31).
MONTH	TINYINT	DATE_ID`s month value (1-12).
YEAR	INT	DATE_ID`s year value (2021,...).
QUARTER	TINYINT	DATE_ID`s quarter value (1, 2, 3, 4).
YYYYDDD	STRING	DATE_ID`s year value concatentated with the day of the year.
DDD	STRING	DATE_ID`s day of the year.
FIRST_DAY_OF_MONTH	DATE	DATE_ID backed up to the first of the month.
FIRST_DAY_OF_QUARTER	DATE	DATE_ID backed up to the first of the quarter.
FIRST_DAY_OF_YEAR	DATE	DATE_ID backed up to the first of the year.
MONTH_NAME	STRING	DATE_ID`s month as propercase name (January,...).
WEEKDAY_NAME	STRING	DATE_ID`s weekday as propercase name (Monday,...).
YYYYQQ	STRING	DATE_ID`s year and quarter as YYYYQQ (202101,...202104).
YYYYMM	STRING	DATE_ID`s year and month as YYYYMM (202101,...,202112).
YYYYMMDD	STRING	DATE_ID as YYYYMMDD (20210101,...,20211231).

DATE_LONG	STRING	DATE_ID as long text (January 1, 2021,...,December 31, 2021).
DATE_SHORT	STRING	DATE_ID as short text (01JAN2021,...,31DEC2021).

For example, for the date `2020-08-10`, the following values are stored in the table `DIM_CALENDAR`:

- ☐ DATE_ID: 2020-08-10
- ☐ DAY: 8
- ☐ MONTH: 10
- ☐ YEAR: 2020
- ☐ QUARTER: 3
- ☐ YYYYDDD: 2020223
- ☐ DDD: 223
- ☐ FIRST_DAY_OF_MONTH: 2020-08-01
- ☐ FIRST_DAY_OF_QUARTER: 2020-07-01
- ☐ FIRST_DAY_OF_YEAR: 2020-01-01
- ☐ MONTH_NAME: August
- ☐ WEEKDAY_NAME: Monday
- ☐ YYYYQQ: 202003
- ☐ YYYYMM: 202008
- ☐ YYYYMMDD: 20200810
- ☐ DATE_LONG: August 10, 2020
- ☐ DATE_SHORT: 10AUG2020

Now, there are several ways to code this, but here's my first attempt. Take note that I make use of the `EXECUTE` Statement to retrieve values back from Impala making use of ImpalaSQL functions such as `DAYOFYEAR()`, `TRUNC()`, and so on. I also use the `EXECUTE` Statement to insert each day's data into `DIM_CALENDAR`. Finally, I compute statistics on the table once all of the data has been loaded into the database.

```
set hplsql.conn.default=impala;
DECLARE

 sSQL STRING; -- GENERIC SQL STRING

 --SQL INSERT TEMPLATE
 sSQL_INSERT_TMPL STRING := "
                            INSERT INTO DIM_CALENDAR
                             VALUES(
                                    DATE '{sDATEID}',
                                    {sDAY},
                                    {sMONTH},
                                    {sYEAR},
                                    {sQUARTER},
                                    '{sYYYYDDD}',
                                    '{sDDD}',
                                    DATE '{sFIRSTDAYOFMONTH}',
                                    DATE '{sFIRSTDAYOFQUARTER}',
                                    DATE '{sFIRSTDAYOFYEAR}',
                                    '{sMONTHNAME}',
                                    '{sDAYNAME}',
                                    '{sYYYYQQ}',
                                    '{sYYYYMM}',
                                    '{sYYYYMMDD}',
                                    '{sDAYLONG}',
                                    '{sDAYSHORT}'
                                   )
                           ";

 sSQL_INSERT STRING; --FINAL INSERT STRING FOR EACH DAY
```

Don't you just love multiline strings?

```
 dBEGDATE DATE := DATE '2021-01-01'; --BEGINNING DATE
 dCURDATE DATE := dBEGDATE;
 dENDDATE DATE := DATE '2021-12-31'; --ENDING DATE
 bSTATE BOOLEAN := TRUE; --WHILE LOOP STATE
 sDATEID STRING;
 sYEAR STRING;
 sMONTH STRING;
 sDAY STRING;
 sYYYYQQ STRING;
 sYYYYMM STRING;
 sYYYMMDD STRING;
 sQUARTER STRING;
 sYYYYDDD STRING;
 sDDD STRING;
 sFIRSTDAYOFMONTH STRING;
 sFIRSTDAYOFQUARTER STRING;
 sFIRSTDAYOFYEAR STRING;
 sMONTHNAME STRING;
 sDAYNAME STRING;
 sDAYLONG STRING;
 sDAYSHORT STRING;

BEGIN

 --CHANGE TO PROD_SCHEMA
 sSQL := "USE PROD_SCHEMA";
 EXECUTE sSQL;

 --DROP THE TABLE DIM_CALENDAR
 sSQL := "DROP TABLE IF EXISTS DIM_CALENDAR PURGE";
 EXECUTE sSQL;

 --CREATE THE TABLE DIM_CALENDAR
 sSQL := "
          CREATE TABLE DIM_CALENDAR(
                                   DATE_ID DATE,
                                   DAY TINYINT,
                                   MONTH TINYINT,
                                   YEAR INT,
                                   QUARTER TINYINT,
                                   YYYYDDD STRING,
                                   DDD STRING,
                                   FIRST_DAY_OF_MONTH DATE,
                                   FIRST_DAY_OF_QUARTER DATE,
                                   FIRST_DAY_OF_YEAR DATE,
                                   MONTH_NAME STRING,
                                   WEEKDAY_NAME STRING,
                                   YYYYQQ STRING,
                                   YYYYMM STRING,
                                   YYYYMMDD STRING,
                                   DATE_LONG STRING,
                                   DATE_SHORT STRING
                                  )
           STORED AS PARQUET
         ";
 EXECUTE sSQL;

 --LOOP AROUND FROM dBEGDATE TO dENDDATE
 WHILE bSTATE LOOP
```

```
--IF THE CURRENT DATE IS THE SAME AS THE END DATE, CHANGE bSTATE TO FALSE.
IF TO_CHAR(dCURDATE) = TO_CHAR(dENDDATE) THEN
 bSTATE := FALSE;
END IF;

/* GATHER THE COLUMNS FOR THE TABLE. */
--DATE_ID IS FORMATTED AS YYYY-MM-DD AUTOMATICALLY BY TO_CHAR().
sDATEID := TO_CHAR(dCURDATE); --YYYY-MM-DD
sDAY := SUBSTR(sDATEID,9,2);
sMONTH := SUBSTR(sDATEID,6,2);
sYEAR := SUBSTR(sDATEID,1,4);

--DETERMINE THE QUARTER FROM sMONTH.
IF sMONTH='01' OR sMONTH='02' OR sMONTH='03' THEN
 sQUARTER := '1';
ELSEIF sMONTH='04' OR sMONTH='05' OR sMONTH='06' THEN
 sQUARTER := '2';
ELSEIF sMONTH='07' OR sMONTH='08' OR sMONTH='09' THEN
 sQUARTER := '3';
ELSEIF sMONTH='10' OR sMONTH='11' OR sMONTH='12' THEN
 sQUARTER := '4';
END IF;

--CREATE sYYYYMM, sYYYYMMDD AND sYYYYQQ FROM THE OTHER VARIABLES.
sYYYYMM := sYEAR || sMONTH;
sYYYYMMDD := sYEAR || sMONTH || sDAY
sYYYYQQ := sYEAR || "0" || sQUARTER;

--CREATE THE JULIAN DAY FROM IMPALA DIRECTLY.
--DAYOFYEAR() DOES NOT RETURN LEADING ZEROES, SO WE PUT THEM IN BELOW.
sSQL := "SELECT DAYOFYEAR(DATE '" || sDATEID || "')";
EXECUTE sSQL INTO sDDD;
IF LENGTH(sDDD)=1 THEN
 sDDD := "00" || sDDD;
ELSEIF LENGTH(sDDD)=2 THEN
 sDDD := "0" || sDDD;
END IF;

--CREATE THE JULIAN DAY WITH THE YEAR PREPENDED.
sYYYYDDD := sYEAR || sDDD;

--CREATE THE FIRST DAY OF MONTH, QUARTER AND YEAR.
sSQL := "
         SELECT TRUNC(DT,'MONTH') AS FIRST_DAY_OF_MTH,
                TRUNC(DT,'Q') AS FIRST_DAY_OF_QTR,
                TRUNC(DT,'YEAR') AS FIRST_DAY_OF_YR,
                MONTHNAME(DT) AS MONTH_NAME,
                DAYNAME(DT) AS DAY_NAME
          FROM (
                SELECT DATE '" || sDATEID || "' AS DT
               ) A
        ";
EXECUTE sSQL INTO sFIRSTDAYOFMONTH,
                  sFIRSTDAYOFQUARTER,
                  sFIRSTDAYOFYEAR,
                  sMONTHNAME,
                  sDAYNAME;

--CREATE LONG FORMAT (Month dd, yyyy) AND SHORT FORMAT (ddMONYYYY) STRINGS.
sDAYLONG := sMONTHNAME || " " || sDAY || ", " || sYEAR;
```

```
  sDAYSHORT := sDAY  || UPPER(SUBSTR(sMONTHNAME,1,3)) || sYEAR

  --CREATE THIS DAY`S INSERT STRING BY UPDATING THE TEMPLATE.
  sSQL_INSERT := REPLACE(sSQL_INSERT_TMPL,'\{sDATEID\}',sDATEID);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sDAY\}',sDAY);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sMONTH\}',sMONTH);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sYEAR\}',sYEAR);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sQUARTER\}',sQUARTER);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sYYYYDDD\}',sYYYYDDD);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sDDD\}',sDDD);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sFIRSTDAYOFMONTH\}',sFIRSTDAYOFMONTH);
  sSQL_INSERT := REPLACE(sSQL_INSERT,
                                '\{sFIRSTDAYOFQUARTER\}',sFIRSTDAYOFQUARTER);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sFIRSTDAYOFYEAR\}',sFIRSTDAYOFYEAR);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sMONTHNAME\}',sMONTHNAME);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sDAYNAME\}',sDAYNAME);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sYYYYQQ\}',sYYYYQQ);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sYYYYMM\}',sYYYYMM);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sYYYYMMDD\}',sYYYYMMDD);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sDAYLONG\}',sDAYLONG);
  sSQL_INSERT := REPLACE(sSQL_INSERT,'\{sDAYSHORT\}',sDAYSHORT);

  --INSERT TODAY`S DATA INTO THE DATABASE.
  EXECUTE sSQL_INSERT;

  --INCREMENT THE DATE BY ONE DAY.
  dCURDATE := dCURDATE + INTERVAL 1 DAY;
 END WHILE;

 --COMPUTE STATS ON THE TABLE.
 sSQL := 'COMPUTE STATS DIM_CALENDAR';
 EXECUTE sSQL;

END;
```

First, the string `sSQL_INSERT_TMPL` contains the `INSERT INTO` syntax used to insert each row of data into the table. Within this template, the text to be replaced by actual values are surrounded by curly braces. For example, the date ID is `{sDATEID}`. Once all of the pieces have been gathered, the variable `sSQL_INSERT` is updated to contain all of the values to be loaded into the table. This is done by using the `REPLACE()` function multiple times. Take note that when using the `REPLACE()` function, both the left and right curly braces had to be *escaped* with a backslash and was purely due to my choice of curly braces to surround the variables: **{**`sDATEID`**}**. As mentioned before, you can just use the string concatenation operators to do a similar thing, but that tends to produce messy code…and we wouldn't want that!

Take note that at the end of the `WHILE` Loop, the variable `dCURDATE` is shifted by `1` day forward using the `INTERVAL` syntax.

Once the table has been loaded with the selected range of days, a `COMPUTE STATS` is run on the table to gather important statistics.

Working with DML/DDL Natively

While using the `EXECUTE` Statement, as shown above, works great, you can achieve a similar result by using DML/DDL natively from within your HPL/SQL program. But, be aware the HPL/SQL performs *on-the-fly conversion* by replacing legacy database data types with appropriate Hive/Impala data types (we discussed this mapping earlier), `NULL`/`NOT NULL` column constraints are removed, any primary key constraints are removed, and so on.

The following statements are available natively within HL/SQL:

- `CREATE TABLE`
- `DROP TABLE`
- `INSERT INTO TABLE`
- `INSERT OVERWRITE TABLE`
- `TRUNCATE TABLE`
- `UPDATE`
- `SELECT`
- `SELECT INTO`
- `USE`

When using the `SELECT` Statement, rows are displayed as output. If you'd prefer to access the results of a query one row at a time, see the section *Working with Cursors* below.

As an example, let's rework our HPL/SQL code from the previous section to create the table `DIM_CALENDAR` using native DML/DDL.

```
set hplsql.conn.default=impala;
DECLARE

 dBEGDATE DATE := DATE '2021-01-01'; --BEGINNING DATE
 dCURDATE DATE := dBEGDATE;
 dENDDATE DATE := DATE '2021-12-31'; --ENDING DATE
 bSTATE BOOLEAN := TRUE; --WHILE LOOP STATE
 sDATEID STRING;
 sYEAR STRING;
 iYEAR INT;
 sMONTH STRING;
 iMONTH TINYINT;
 sDAY STRING;
 iDAY TINYINT;
 sYYYYQQ STRING;
 sYYYYMM STRING;
 sYYYMMDD STRING;
 sQUARTER STRING;
 iQUARTER TINYINT;
 sYYYYDDD STRING;
 sDDD STRING;
 sFIRSTDAYOFMONTH STRING;
 dFIRSTDAYOFMONTH DATE;
 sFIRSTDAYOFQUARTER STRING;
 sFIRSTDAYOFYEAR STRING;
 sMONTHNAME STRING;
 sDAYNAME STRING;
 sDAYLONG STRING;
 sDAYSHORT STRING;

BEGIN

 --CHANGE TO PROD_SCHEMA
 USE PROD_SCHEMA;

 --DROP THE TABLE DIM_CALENDAR
 DROP TABLE IF EXISTS DIM_CALENDAR PURGE;

 --CREATE THE TABLE DIM_CALENDAR
 CREATE TABLE DIM_CALENDAR(
                          DATE_ID DATE,
```

```
                                DAY TINYINT,
                                MONTH TINYINT,
                                YEAR INT,
                                QUARTER TINYINT,
                                YYYYDDD STRING,
                                DDD STRING,
                                FIRST_DAY_OF_MONTH DATE,
                                FIRST_DAY_OF_QUARTER DATE,
                                FIRST_DAY_OF_YEAR DATE,
                                MONTH_NAME STRING,
                                WEEKDAY_NAME STRING,
                                YYYYQQ STRING,
                                YYYYMM STRING,
                                YYYYMMDD STRING,
                                DATE_LONG STRING,
                                DATE_SHORT STRING
                               )
 STORED AS PARQUET;

--LOOP AROUND FROM dBEGDATE TO dENDDATE
WHILE bSTATE LOOP

 --IF THE CURRENT DATE IS THE SAME AS THE END DATE, CHANGE bSTATE TO FALSE.
 IF TO_CHAR(dCURDATE) = TO_CHAR(dENDDATE) THEN
  bSTATE := FALSE;
 END IF;

 /* GATHER THE COLUMNS FOR THE TABLE. */
 --DATE_ID IS FORMATTED AS YYYY-MM-DD AUTOMATICALLY BY TO_CHAR().
 sDATEID := TO_CHAR(dCURDATE); --YYYY-MM-DD
 sDAY := SUBSTR(sDATEID,9,2);
 iDAY := CAST(SUBSTR(sDATEID,9,2) AS TINYINT);
 sMONTH := SUBSTR(sDATEID,6,2);
 iMONTH := CAST(SUBSTR(sDATEID,6,2) AS TINYINT);
 sYEAR := SUBSTR(sDATEID,1,4);
 iYEAR := CAST(SUBSTR(sDATEID,1,4) AS INT);

 --DETERMINE THE QUARTER FROM sMONTH.
 IF sMONTH='01' OR sMONTH='02' OR sMONTH='03' THEN
  sQUARTER := '1';
 ELSEIF sMONTH='04' OR sMONTH='05' OR sMONTH='06' THEN
  sQUARTER := '2';
 ELSEIF sMONTH='07' OR sMONTH='08' OR sMONTH='09' THEN
  sQUARTER := '3';
 ELSEIF sMONTH='10' OR sMONTH='11' OR sMONTH='12' THEN
  sQUARTER := '4';
 END IF;
 iQUARTER := CAST(sQUARTER AS TINYINT);

 --CREATE sYYYYMM, sYYYYMMDD AND sYYYYQQ FROM THE OTHER VARIABLES.
 sYYYYMM := sYEAR || sMONTH;
 sYYYYMMDD := sYEAR || sMONTH || sDAY
 sYYYYQQ := sYEAR || "0" || sQUARTER;

 --CREATE THE JULIAN DAY FROM IMPALA DIRECTLY.
 --DAYOFYEAR() DOES NOT RETURN LEADING ZEROES, SO WE PUT THEM IN BELOW.
 SELECT DAYOFYEAR(sDATEID)
  INTO sDDD;

 IF LENGTH(sDDD)=1 THEN
```

```
  sDDD := "00" || sDDD;
 ELSEIF LENGTH(sDDD)=2 THEN
  sDDD := "0" || sDDD;
 END IF;

 --CREATE THE JULIAN DAY WITH THE YEAR PREPENDED.
 sYYYYDDD := sYEAR || sDDD;

 --CREATE THE FIRST DAY OF MONTH, QUARTER AND YEAR.
SELECT CAST(TRUNC(DT,'MONTH') AS STRING),
       CAST(TRUNC(DT,'Q') AS STRING),
       CAST(TRUNC(DT,'YEAR') AS STRING),
       MONTHNAME(DT),
       DAYNAME(DT)
 INTO sFIRSTDAYOFMONTH,sFIRSTDAYOFQUARTER,sFIRSTDAYOFYEAR,sMONTHNAME,sDAYNAME
 FROM (
       SELECT sDATEID AS DT
      ) A;

 --CREATE LONG FORMAT (Month dd, yyyy) AND SHORT FORMAT (ddMONYYYY) STRINGS.
 sDAYLONG := sMONTHNAME || " " || sDAY || ", " || sYEAR;
 sDAYSHORT := sDAY  || UPPER(SUBSTR(sMONTHNAME,1,3)) || sYEAR

 --INSERT THIS DAY`S INFO INTO THE TABLE.
 INSERT INTO DIM_CALENDAR
  SELECT sDATEID,
         iDAY,
         iMONTH,
         iYEAR,
         iQUARTER,
         sYYYYDDD,
         sDDD,
         sFIRSTDAYOFMONTH,
         sFIRSTDAYOFQUARTER,
         sFIRSTDAYOFYEAR,
         sMONTHNAME,
         sDAYNAME,
         sYYYYQQ,
         sYYYYMM,
         sYYYYMMDD,
         sDAYLONG,
         sDAYSHORT;

 --INCREMENT THE DATE BY ONE DAY.
 dCURDATE := dCURDATE + INTERVAL 1 DAY;
END WHILE;

--COMPUTE STATS ON THE TABLE.
sSQL := 'COMPUTE STATS DIM_CALENDAR';
EXECUTE sSQL;

END;
```

As you can see in the code above, the `EXECUTE` Statements have been replaced with native code. Since the `INSERT INTO` expects the appropriate data types, the variables `sDAY`, `sMONTH` and `sQUARTER` have corresponding doppelgangers `iDAY`, `iMONTH` and `iQUARTER` set to `TINYINT`; and the variable `sYEAR` has `iYEAR` set to `INT`. Although we're passing `sDATEID` as a `STRING`, since `DATE_ID` is defined as a `DATE` data type in the table, ImpalaSQL will convert it automatically.

Note that the following SQL code makes use of the ImpalaSQL function `DAYOFYEAR()`. This SQL code is passed by HPL/SQL to the database to compute the value…

```
SELECT DAYOFYEAR(sDATEID)
   INTO sDDD;
```

…but the following code needed to be jiggered a bit…

```
SELECT CAST(TRUNC(DT,'MONTH') AS STRING),
       CAST(TRUNC(DT,'Q') AS STRING),
       CAST(TRUNC(DT,'YEAR') AS STRING),
       MONTHNAME(DT),
       DAYNAME(DT)
 INTO sFIRSTDAYOFMONTH,sFIRSTDAYOFQUARTER,sFIRSTDAYOFYEAR,sMONTHNAME,sDAYNAME
 FROM (
       SELECT sDATEID AS DT
      ) A;
```

This jiggering was due to the nesting of the functions `CAST` and `TRUNC` causing HPL/SQL some trouble. This sucks royal moose, and the workaround is to make a subquery for the `sDATEID`, as shown above. Without the subquery, shown below, HPL/SQL indicates that it cannot find `sDATEID`.

```
 SELECT CAST(TRUNC(sDATEID,'MONTH') AS STRING),
        CAST(TRUNC(sDATEID,'Q') AS STRING),
        CAST(TRUNC(sDATEID,'YEAR') AS STRING),
        MONTHNAME(sDATEID),
        DAYNAME(sDATEID)
  INTO sFIRSTDAYOFMONTH,sFIRSTDAYOFQUARTER,sFIRSTDAYOFYEAR,sMONTHNAME,sDAYNAME;
```

```
java.sql.SQLException: [Cloudera][ImpalaJDBCDriver](500051) ERROR processing
query/statement. Error Code: 0, SQL state: TStatus(statusCode:ERROR_STATUS,
sqlState:HY000, errorMessage:AnalysisException: Could not resolve column/field
reference: 'sdateid'
), Query: SELECT CAST(TRUNC(sDATEID,'MONTH') AS STRING),
CAST(TRUNC(sDATEID,'Q') AS STRING), CAST(TRUNC(sDATEID,'YEAR') AS STRING),
MONTHNAME('2021-01-01'), DAYNAME('2021-01-01').
```

Note that `sMONTHNAME` and `sDAYNAME` are receiving the correct values since these two columns don't use nested functions. In any case, HPL/SQL passes this SQL code to Impala to be evaluated.

Using `DBMS_OUTPUT` and `PRINT` to Create External Files

As indicated earlier in the book, when using multiple `INSERT INTO` Statements, each row is associated with a separate file in HDFS. For example, here's a partial listing for the table `DIM_CALENDAR`:

```
[smithbob@lnxserver ~]$ hadoop fs -ls -R
                            hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_calendar

-rw-rw----+  3 impala hive       4685 2022-03-24 10:59
hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_calendar/delta_10_10/b44f1b8a027018
36-af00f19300000000_1527263800_data.0.parq

-rw-rw----+  3 impala hive       4685 2022-03-24 10:59
hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_calendar/delta_11_11/ba46de5e33074e
11-4031dafb00000000_1033013878_data.0.parq

-rw-rw----+  3 impala hive       4692 2022-03-24 10:59
hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_calendar/delta_12_12/dd45f1ce44adf3
8d-8a1ce48700000000_1619420159_data.0.parq
```

```
-rw-rw----+  3 impala hive       4706 2022-03-24 10:59
hdfs://lnxserver.com:8020/warehouse/tablespace/managed/hive/dim_calendar/delta_13_13/554f1cfd17af17
62-ef007dbf00000000_450424858_data.0.parq
```

There are `31 .parq` files below the HDFS `dim_calendar` directory, one corresponding to each day inserted into the table `DIM_CALENDAR` (testing just one month, January 2021, here). How can we live like this!?! One way around this is to create a text file containing delimited data which will be used to load data into the table `DIM_CALENDAR`. The package `DBMS_OUTPUT` as well as the `PRINT()` function can be used to create external files with the use of the handy-dandy redirection arrow (`>`). It doesn't matter which one you choose, so take your pick, kids. The `DBMS_OUTPUT` package contains the following function:

DBMS_OUTPUT PACKAGE		
Function	**Return Type**	**Description**
PUT_LINE(*sText*)	N/A	Prints ***sText*** to the `STDOUT`. A line terminator is added to the end of the line automatically.

Let's rewrite our HPL/SQL code using `DBMS_OUTPUT` to create a semicolon delimited file containing the `DIM_CALENDAR` data. First, we can remove the HPL/SQL variables `iDAY`, `iMONTH`, `iQUARTER` and `iYEAR` since the corresponding string variables will be used instead. Next, we declare a string variable to hold the entire contents of the delimited line:

```
--A STRING VARIABLE TO HOLD AN ENTIRE LINE OF DATA.
sLINE STRING;
```

Near the end of the loop, the day's line of data is written to the file:

```
--CREATE THIS DAY`S INSERT STRING.
sLINE := sDATEID || ";" ||
         sDAY || ";" ||
         sMONTH || ";" ||
         sYEAR || ";" ||
         sQUARTER || ";" ||
         sYYYYDDD || ";" ||
         sDDD || ";" ||
         SUBSTR(sFIRSTDAYOFMONTH,1,10) || ";" ||
         SUBSTR(sFIRSTDAYOFQUARTER,1,10) || ";" ||
         SUBSTR(sFIRSTDAYOFYEAR,1,10) || ";" ||
         sMONTHNAME || ";" ||
         sDAYNAME || ";" ||
         sYYYYQQ || ";" ||
         sYYYYMM || ";" ||
         sYYYYMMDD || ";" ||
         sDAYLONG || ";" ||
         sDAYSHORT;

--WRITE THE LINE TO THE FILE
DBMS_OUTPUT.PUT_LINE(sLINE);
```

After running our newly updated HPL/SQL code…

```
[smithbob@lnxserver ~]$ hplsql -f dimcalendar.hplsql > tmp_calendar.dlm
```

…we can take a look-see at some of the rows of data in `tmp_calendar.dlm`…

```
2021-01-01;01;01;2021;1;2021001;001;2021-01-01 00:00:00;2021-01-01 00:00:00;2021-01-01 00:00:00;January;...snip...
2021-01-02;02;01;2021;1;2021002;002;2021-01-01 00:00:00;2021-01-01 00:00:00;2021-01-01 00:00:00;January;...snip...
2021-01-03;03;01;2021;1;2021003;003;2021-01-01 00:00:00;2021-01-01 00:00:00;2021-01-01 00:00:00;January;...snip...
2021-01-04;04;01;2021;1;2021004;004;2021-01-01 00:00:00;2021-01-01 00:00:00;2021-01-01 00:00:00;January;...snip...
2021-01-05;05;01;2021;1;2021005;005;2021-01-01 00:00:00;2021-01-01 00:00:00;2021-01-01 00:00:00;January;...snip...
...snip...
```

We can copy this file to a directory in HDFS named `tmp_calendar`...

```
[smithbob@lnxserver ~]$ hadoop fs -mkdir
           hdfs://lnxserver.com:8020/warehouse/tablespace/external/tmp_calendar

[smithbob@lnxserver ~]$ hadoop fs -put tmp_calendar.dlm
           hdfs://lnxserver.com:8020/warehouse/tablespace/external/tmp_calendar
```

and using ImpalaSQL, we can create our final managed table `DIM_CALENDAR` in `PARQUET` format casting to the appropriate data types:

```
drop table dim_calendar purge;
create table dim_calendar(
 date_id date,
 day tinyint,
 month tinyint,
 year int,
 quarter tinyint,
 yyyyddd string,
 ddd string,
 first_day_of_month date,
 first_day_of_quarter date,
 first_day_of_year date,
 month_name string,
 weekday_name string,
 yyyyqq string,
 yyyymm string,
 yyyymmdd string,
 date_long string,
 date_short string
)
 stored as parquet;

insert into dim_calendar
 select cast(date_id as date format 'YYYY-MM-DD'),
        cast(day as tinyint),
        cast(month as tinyint),
        cast(year as int),
        cast(quarter as tinyint),
        yyyyddd,
        ddd,
        cast(first_day_of_month as date format 'YYYY-MM-DD'),
        cast(first_day_of_quarter as date format 'YYYY-MM-DD'),
        cast(first_day_of_year as date format 'YYYY-MM-DD'),
        month_name,
        weekday_name,
        yyyyqq,
        yyyymm,
        yyyymmdd,
        date_long,
        date_short
  from tmp_calendar;

compute stats dim_calendar;

drop table tmp_calendar purge;
```

Sweet!

Using `UTL_FILE` to Create Files Directly in HDFS

As indicated in the previous section, when using multiple `INSERT INTO` Statements, each row is associated with a separate file in HDFS, but one way around this was to create a text file containing delimited data used to load data into the table `TMP_CALENDAR` to create the final table `DIM_CALENDAR`. On the other hand, the package `UTL_FILE` is used to **create files in HDFS directly**. The `UTL_FILE` package contains the following bangin' functions and a lone type:

UTL_FILE FUNCTIONS AND THE LONE TYPE		
Type	**Return Type**	**Description**
UTL_FILE.FILE_TYPE	UTL_FILE.FILE_ TYPE	File handle used with the functions listed below.
Functions	**Return Type**	**Description**
FOPEN(*directory*,*filename*,*mode*)	UTL_FILE.FILE_ TYPE	Returns ***UTL_FILE.FILE_TYPE*** object (referred to as ***oFile*** below) pointing to the HDFS ***directory***/***filename***. The ***mode*** can be `r` for read, `w` for write. If the ***filename*** doesn't exist, it's created; otherwise, it's overwritten.
GET_LINE(*oFile*, *hplsql-var*, *max-line-length*)	N/A	Reads a single row from the file associated with ***oFile*** and stores it in ***hplsql-var***. If ***max-line-length*** is specified, the row is truncated to this length.
PUT_LINE(*oFile*,*sText*)	N/A	Inserts ***sText*** into the file associated with ***oFile***. A line terminator is added to the end of the line automatically.
PUT(*oFile*,*sText*)	N/A	Inserts ***sText*** into the file associated with ***oFile***. A line terminator is **NOT** added to the end of the line.
FCLOSE(*oFile*)	N/A	Closes the file associated with ***oFile***.

Let's rewrite our HPL/SQL code using `UTL_FILE` to create a delimited file containing the `DIM_CALENDAR` data. First, we can delete the HPL/SQL variables `iDAY`, `iMONTH`, `iQUARTER` and `iYEAR` since the corresponding string variables will be used instead. Next, we declare a string variable to hold the entire contents of the delimited line:

```
--A STRING VARIABLE TO HOLD AN ENTIRE LINE OF DATA.
sLINE STRING;
```

Next, we declare the file object, `oFILE`:

```
--FILE HANDLE FOR EXTERNAL FILE.
oFILE UTL_FILE.FILE_TYPE;
```

Inside the `BEGIN` section, we open a file named `dim_calendar.dlm` for write in HDFS:

```
--OPEN THE EXTERNAL HDFS FILE.
oFILE=UTL_FILE.FOPEN('hdfs://lnxserver.com:8020/warehouse/tablespace/
                            managed/hive/tmp_calendar','tmp_calendar.dlm','w');
```

Near the end of the loop, the day's line of data is written to the file:

```
  --CREATE THIS DAY`S INSERT STRING.
  sLINE := sDATEID || ";" ||
           sDAY || ";" ||
           sMONTH || ";" ||
           sYEAR || ";" ||
           sQUARTER || ";" ||
           sYYYYDDD || ";" ||
           sDDD || ";" ||
           sFIRSTDAYOFMONTH || ";" ||
           sFIRSTDAYOFQUARTER || ";" ||
           sFIRSTDAYOFYEAR || ";" ||
           sMONTHNAME || ";" ||
           sDAYNAME || ";" ||
```

```
            sYYYYQQ || ";" ||
            sYYYYMM || ";" ||
            sYYYYMMDD || ";" ||
            sDAYLONG || ";" ||
            sDAYSHORT;

    --WRITE THE LINE TO THE FILE
    UTL_FILE.PUT_LINE(oFILE,sLINE);
```

Finally, the file is closed:

```
--CLOSE THE FILE
UTL_FILE.FCLOSE(oFILE);
```

At this point, we have a file named `tmp_calendar.dlm` located in the `tmp_calendar` folder in HDFS. Naturally, we have to tell ImpalaSQL that this table exists:

```
drop table tmp_calendar;
create external table tmp_calendar(
 date_id string,
 day string,
 month string,
 year string,
 quarter string,
 yyyyddd string,
 ddd string,
 first_day_of_month string,
 first_day_of_quarter string,
 first_day_of_year string,
 month_name string,
 weekday_name string,
 yyyyqq string,
 yyyymm string,
 yyyymmdd string,
 date_long string,
 date_short string
)
 row format delimited
 fields terminated by ';'
 stored as textfile
 location 'hdfs://lnxserver.com:8020/warehouse/tablespace/
                                                              external/tmp_calendar'
 tblproperties('skip.header.line.count'='0');
```

Next, let's create our final `DIM_CALENDAR` table:

```
drop table dim_calendar purge;
create table dim_calendar(
 date_id date,
 day tinyint,
 month tinyint,
 year int,
 quarter tinyint,
 yyyyddd string,
 ddd string,
 first_day_of_month date,
 first_day_of_quarter date,
 first_day_of_year date,
 month_name string,
 weekday_name string,
```

```
 yyyyqq string,
 yyyymm string,
 yyyymmdd string,
 date_long string,
 date_short string
)
 stored as parquet;
```

And, finally, let's insert the data from `TMP_CALENDAR` into `DIM_CALENDAR`. Here's where there's an issue: The function `UTL_FILE.PUT_LINE()` writes a null character (`x'00'`) in front of each character written to the file `tmp_calendar.dlm`. Selecting from `TMP_CALENDAR` from ImpalaSQL will show that everything is fine, but when you attempt to `CAST()` the columns you'll run into trouble. One quick workaround is to use the `REGEXP_REPLACE()` function to remove the null characters. Here's an example on the `DATE_ID` column:

```
REGEXP_REPLACE(DATE_ID,'\\x00','')
```

With that said, here's our `INSERT` Statement:

```
insert into dim_calendar
 select cast(regexp_replace(date_id,'\\x00','') as date format 'YYYY-MM-DD'),
        cast(regexp_replace(day,'\\x00','') as tinyint),
        cast(regexp_replace(month,'\\x00','') as tinyint),
        cast(regexp_replace(year,'\\x00','') as int),
        cast(regexp_replace(quarter,'\\x00','') as tinyint),
        regexp_replace(yyyyddd,'\\x00',''),
        regexp_replace(ddd,'\\x00',''),
        cast(substr(regexp_replace(first_day_of_month,'\\x00',''),1,10)
                                          as date format 'YYYY-MM-DD'),
        cast(substr(regexp_replace(first_day_of_quarter,'\\x00',''),1,10)
                                          as date format 'YYYY-MM-DD'),
        cast(substr(regexp_replace(first_day_of_year,'\\x00',''),1,10)
                                          as date format 'YYYY-MM-DD'),
        regexp_replace(month_name,'\\x00',''),
        regexp_replace(weekday_name,'\\x00',''),
        regexp_replace(yyyyqq,'\\x00',''),
        regexp_replace(yyyymm,'\\x00',''),
        regexp_replace(yyyymmdd,'\\x00',''),
        regexp_replace(date_long,'\\x00',''),
        regexp_replace(date_short,'\\x00','')
  from tmp_calendar;

compute stats dim_calendar;
drop table tmp_calendar purge;
```

Nice!!

Working with Cursors

Using the `EXECUTE` Statement to retrieve one or more values from a `SELECT` Statement is great, until it's not: just like a relationship that's slowly going down the toilet. In that case, you can use a cursor to process the results of a query one row at a time. There are several ways to create a cursor, but the easiest is an extension of the `FOR` Statement:

```
FOR cursor-name IN (select-statement) LOOP

...statements...

END LOOP;
```

Within the loop, you can access the columns of the *select-statement* with the following dot syntax:

```
cursor-name.column-name
```

For example, let's display the columns POSTAL_CODE and CITY from the table DIM_POSTAL_CODE for the state of New Jersey:

```
set hplsql.conn.default=impala;
DECLARE

 sSQL string;

BEGIN

--Change to PROD_SCHEMA
USE PROD_SCHEMA;

 FOR csrNJONLY IN (
                  SELECT POSTAL_CODE,CITY
                   FROM DIM_POSTAL_CODE
                   WHERE STATE_CODE='NJ'
                   ORDER BY POSTAL_CODE
                 ) LOOP

  PRINT csrNJONLY.POSTAL_CODE + "/" + csrNJONLY.CITY;

 END LOOP;

END;
```

And the results are…

```
07001/AVENEL
07002/BAYONNE
07003/BLOOMFIELD
07004/FAIRFIELD
07005/BOONTON
07006/CALDWELL
07007/CALDWELL
07008/CARTERET
07009/CEDAR GROVE
07010/CLIFFSIDE PARK
07011/CLIFTON
...snip...
```

Take note that the SQL query doesn't need to be enclosed in quotes. Also, note that both columns are accessed using the dot syntax csrNJONLY.POSTAL_CODE and csrNJONLY.CITY.

Another method to create a cursor is with DECLARE CURSOR Statement. This method is slightly more complicated than the FOR Statement for cursors because you'll have to perform the following tasks in order to use it:

1. Create the cursor using the DECLARE CURSOR Statement
2. Open the cursor using the OPEN Statement
3. Fetch the current row's column values using the FETCH Statement
4. Repeat Step #3 using a WHILE Statement
5. Close the cursor using the CLOSE Statement

Now, before you run away screaming into the darkness of night, it's really not that bad. Here's the general syntax:

```
--Declare the cursor
DECLARE cursor-name CURSOR FOR select-statement;

--Open the cursor
OPEN cursor-name;

--Fetch the first row`s data into the HPL/SQL variable(s)
FETCH cursor-name INTO hplsql-var1,hplsql-var2,...

--Process the remaining rows one row at a time
WHILE SQLCODE=0 LOOP

 ...statements...

 --Fetch the next row
 FETCH cursor-name INTO hplsql-var1,hplsql-var2,...
END WHILE

--All done?  Be a good citizen and close the cursor!
CLOSE cursor-name
```

Take note of the automatic built-in variable `SQLCODE`. This variable remains zero until there are no more rows in the query results, in which case it will be set to `100`. Why `100`? Who the hell knows.

For example, let's redo the previous example using the `DECLARE CURSOR` Statement and attendant friends:

```
set hplsql.conn.default=impala;
DECLARE

 sSQL string;
 sPOSTALCODE string;
 sCITY string;

BEGIN

 --Change to PROD_SCHEMA
 USE PROD_SCHEMA;

 --SQL Query to pull only NJ from the DIM_POSTAL_CODE table.
 sSQL := "
          SELECT POSTAL_CODE,CITY
           FROM DIM_POSTAL_CODE
           WHERE STATE_CODE='NJ'
           ORDER BY POSTAL_CODE
         ";

 --Declare the cursor.
 DECLARE csrNJONLY CURSOR FOR sSQL;

 --Open the cursor.
 OPEN csrNJONLY;

 --Fetch the first row`s data.
 FETCH csrNJONLY INTO sPOSTALCODE,sCITY;

 --Loop around for each of the remaining rows of data.
 WHILE SQLCODE=0 LOOP

  PRINT sPOSTALCODE + "/" + sCITY;
```

```
   --Don`t forget the fetch inside the loop!!
   FETCH csrNJONLY INTO sPOSTALCODE,sCITY;
  END WHILE;

  --Close the cursor.
  CLOSE csrNJONLY;

END;
```

Finally, there are two additional cursor-related statements: `ALLOCATE CURSOR` and `ASSOCIATE LOCATOR`. These statements allow you to use a cursor with rows returned directly from a stored procedure. If that's your pleasure, please see the HPL/SQL online documentation for more on these two statements.

Yoo-Hoo! Calling the Operating System!

Occasionally, you may need to run a Linux command from within your HPL/SQL program whether to create a directory, remove a file, send an email, etc. You can do this by using the `HOST` Statement:

```
HOST 'command-string';
```

For example, let's run an ImpalaSQL command from within an HPL/SQL program to gather statistics on the table `DIM_POSTAL_CODE`:

```
DECLARE

 sCMD string;

BEGIN

 sCMD := "impala-shell -q 'use prod_schema;compute stats dim_postal_code;'";
 HOST sCMD;

END;
```

When run at the command line, this is what you'll see:

```
Error, could not parse arguments "prod_schema; compute stats dim_postal_code;'"
```

Well, that's not good! In fact, for any remotely complicated command line given to `HOST`, you'll receive an error. The way around this is to create a Linux script which accepts a few simple parameters. For example, to perform `COMPUTE STATS` on a requested table in `prod_schema`, let's create a Linux script called `impalaTableStats` to do just that:

```
#!/bin/bash

# Schema name
echo $1

# Table name
echo $2

# Form the command to run...your command may be more complicated.
sComputeStats="impala-shell --query='use $1;compute stats $2;'"
echo $sComputeStats

# Run the command.
eval $sComputeStats
```

```
exit
```

Next, let's create an HPL/SQL procedure called `IMPALA_STATS()`, located in the file `impala_stats.hplsql`, which accepts a schema and table name and then calls the Linux script above:

```
CREATE OR REPLACE PROCEDURE IMPALA_STATS(psSCHEMA IN STRING,
                                         psTABLENAME IN STRING) AS

 sSQL STRING;

BEGIN

 sSQL := '/directory/impalaTableStats ' || psSCHEMA || ' ' || psTABLENAME;
 PRINT sSQL;
 HOST sSQL;

END;
```

To execute this procedure, include it in your program and call it, like this:

```
set hplsql.conn.default=impala;
INCLUDE impala_stats.hplsql
DECLARE

 sSCHEMA STRING;
 sTABLE STRING;

BEGIN

 sSCHEMA := "PROD_SCHEMA";
 sTABLE := "DIM_POSTAL_CODE";
 CALL IMPALA_STATS(sSCHEMA,sTABLE);

END;
```

So, in order to use `HOST`, the name of the game is clean and simple! For more complicated parameters, you may need to surround them individually by quotes so your Linux script assigns `$1`, `$2`, etc. to your parameters correctly. If your parameters themselves contain apostrophes or doublequotes, I tend to replace them by a set of characters that don't usually appear together. For example, I tend to replace apostrophes by **`{<>}`**, doublequotes by **`{()}`**, spaces by **`{[]}`**, and so on. Overkill? Sure! I then surround each parameter by doublequotes to ensure that the Linux script will pick them up properly. Then, within the Linux script, I do the reverse process. *A programmer's gotta do what a programmer's gotta do!*

Now, the `HOST` Statement allows you to retrieve a return code from either a native command or one produced by your own script. This return code is placed in the HPL/SQL built-in variable `HOSTCODE` and should be captured **directly after** the `HOST` Statement. For example, within the procedure above, we can capture `HOSTCODE` and print out a message if something went balls up:

```
CREATE OR REPLACE PROCEDURE IMPALA_STATS(psSCHEMA IN STRING,
                                         psTABLENAME IN STRING) AS

 sSQL STRING;

BEGIN

 sSQL := '/directory/impalaTableStats ' || psSCHEMA || ' ' || psTABLENAME;
 PRINT sSQL;
 HOST sSQL;
```

```
 /* CAPTURE THE RETURN CODE FROM THE HOST STATEMENT */
 IF HOSTCODE <> 0 THEN

  PRINT "ERROR: HOST COMMAND >>> " + sSQL + " <<< FAILED WITH RETURN CODE " +
TO_CHAR(HOSTCODE) + "! PLEASE HELP! I'M LOST!";

 END IF;

END;
```

To test this out, a slight change was made to the Linux script `impalaTableStats`. The keyword `exit` was replaced with `exit 1` to force a non-zero return code back to the caller: the `HOST` Statement in this case. The built-in variable `HOSTCODE` will then be set to `1` and the `IF` Statement shown above will execute:

```
ERROR: HOST COMMAND >>> /directory/impalaTableStats PROD_SCHEMA DIM_POSTAL_CODE
<<< FAILED WITH RETURN CODE 1! HELP! I'M TRAPPED IN A FORTUNE COOKIE FACTORY
AND CAN'T GET OUT!
```

Recall that the `exit` command will exit with the return code of the last command executed in the script. By forcing the script to exit with a return code of `1`, we were able to test `HOSTCODE` above.

Chapter 28 – Handling HPL/SQL Exceptions

In this section, we explore how to intercept and handle exceptions in an HPL/SQL program. HPL/SQL allows for the traditional `EXCEPTION` Statement, but also has several built-in variables and the ability to declare your own error handlers.

Built-In Variables

In the last chapter, we discussed how to capture the `HOSTCODE` built-in variable following the `HOST` Statement. If `HOSTCODE` contains a non-zero value, something probably went horribly wrong. We also used the `SQLCODE` built-in variable with cursors to tell a `WHILE` Statement when to stop looping.

Now, the built-in variable `SQLSTATE` contains the return code from the last SQL statement issued, whether via the `EXECUTE` Statement or by using native DML/DDL. The built-in variable `SQLSTATE` is a **string** of `5` numeric values and a few of the values you may receive are as follows:

1. `00000` – Successful execution of your SQL code. Note that the built-in variable `SQLCODE` will be set to `0` as well.
2. `01000` – No more data available and corresponds to `SQLCODE` being set to `100`.
3. `02000` – A SQL error occurred and corresponds to `SQLCODE` being set to `-1`.

On any given happy-fluffy day, `SQLSTATE` will be `00000` indicating everything went swimmingly well. Huzzah!!

The `EXCEPTION` Statement and `HPLSQL.ONERROR` Configuration Options

Recall earlier I described the generic form of an HPL/SQL program involving the `DECLARE`, `BEGIN` and `END` Statements. In fact, you can add the `EXCEPTION` Statement which is used to capture one or more errors. So, the generic form now looks like this:

```
...statements...
DECLARE

 ...statements...

BEGIN

 ...statements...

EXCEPTION

 WHEN condition-1 THEN

  ...statements...

 WHEN condition-2 THEN

  ...statements...

...

 WHEN condition-n THEN

  ...statements...

 WHEN OTHERS THEN
```

```
   ...statements...

END;
```

Unfortunately, unlike Oracle, there are no built-in error conditions, such as `ZERO_DIVIDE`, `TOO_MANY_ROWS`, `NO_DATA_FOUND`, etc. you can slap in for *`condition-i`*, so the generic syntax above really reduces down to the following:

```
...statements...
DECLARE

 ...statements...

BEGIN

 ...statements...

EXCEPTION

 WHEN OTHERS THEN

   ...statements...

END;
```

For example, the classic *oopsey!* in programming is dividing by zero. Below is some code to cause this heinous and debilitating error:

```
DECLARE

 dVAL double := -999.0;
 dNUM int := 5;
 dDEN int := 0;

BEGIN

 --DIVIDE dNUM BY dDEN...WHAT COULD POSSIBLY GO WORNG?
 dVAL := dNUM / dDEN;
 PRINT dVAL;

EXCEPTION
 WHEN OTHERS THEN
  PRINT ">>>>> A BIG HONKIN' ERROR OCCURRED!! <<<<<";

END;
```

When this HPL/SQL program is run at the Linux command line, the following message is output:

```
>>>>> A BIG HONKIN' ERROR OCCURRED!! <<<<<
```

Take note that the `PRINT dVAL;` code is never reached; once the error occurs, the `WHEN OTHERS` Condition is triggered. But, this particular behavior occurs because of the default setting of HPL/SQL's `ONERROR` Configuration Option. You can change this option using the following syntax:

```
SET HPLSQL.ONERROR = configuration-option;
```

where the configuration options are as follows:

- ☐ `EXCEPTION` – If an error occurs, an exception is raised. How this is handled depends on whether you code an `EXCEPTION WHEN OTHERS` condition or you create your own condition handler. We discuss condition handlers later on in this chapter. (This option is the default behavior.)
- ☐ `SETERROR` – If an error occurs, the built-in variables `SQLCODE`, `ERRORCODE` and `HOSTCODE` (if applicable) are set and the execution continues with the next statement. You can capture the codes from these built-in variables using an `IF` Statement.
- ☐ `STOP` – If an error occurs, HPL/SQL stops executing the program and exits.

For example, let's update the divide by zero nightmare above by setting the configuration option to `SETERROR`:

```
set hplsql.onerror=seterror;
DECLARE

 dVAL double := -999.0;
 dNUM int := 5;
 dDEN int := 0;

BEGIN

 --DIVIDE dNUM BY dDEN...WHAT COULD POSSIBLY GO WORNG?
 dVAL := dNUM / dDEN;
 PRINT dVAL;

EXCEPTION
 WHEN OTHERS THEN
  PRINT ">>>>> A BIG HONKIN' ERROR OCCURRED!! <<<<<";

END;
```

When this program is executed, the output is as follows:

```
-999.0
```

Although the error still occurs, the error message is not printed out, and the code continues on to the `PRINT dVAL;` code which is why `-999.0` appears above.

Now, if we replace `SETERROR` in `HPLSQL.ONERROR` with the `STOP` configuration option, the program stops at the point where the error occurs, the `PRINT dVAL;` code is never reached, and the error message is not printed. The program literally and figuratively drops dead at the point where the error occurs.

Although `EXCEPTION` is very limited, HPL/SQL allows you to declare your own exceptions and raise them when a nefarious condition is met. We talk about this in the next section.

Creating User-Defined Conditions

In order to declare your own user-defined conditions, you make use of the following statements in roughly the following order:

- ☐ `DECLARE CONDITION` Statement – Use this statement to declare a user-defined condition. You use the `SIGNAL` Statement as a flare gun which shoots a flare up getting the attention of the associated handler causing its code to execute.
- ☐ `DECLARE HANDLER` Statement – The code to be executed when the flare is noticed. *Huh? What's that in the sky?*
- ☐ `SIGNAL` Statement – The flare gun.

The `DECLARE CONDITION` Statement takes the following simple syntax:

```
DECLARE user-defined-condition-name CONDITION;
```

The `DECLARE HANDLER` Statement takes on the following syntax:

```
DECLARE control-option HANDLER FOR handler-option
BEGIN

 ...statements...

END;
```

There are two *control-option*s to choose from:

- `CONTINUE` – This option indicates that control is returned to the statement following the statement that raised the condition.
- `EXIT` – This option indicates that once the handler's code has completed, control is returned to the end of the block which encloses the `DECLARE HANDLER` Statement. Usually, this is the end of the HPL/SQL program causing the program to exit.

There are three *handler-option*s to choose from:

- *user-defined-condition-name* – This option indicates that the handler has been specifically written for a user-defined condition.
- `SQLEXCEPTION` – This option indicates that the handler has been specifically written to handle SQL exceptions.
- `NOT FOUND` – This option indicates that the handler has been specifically written to handle `NOT FOUND` exceptions.

For a user-defined condition which just prints out an error message and then causes the program to drop dead, you can code something like this:

```
DECLARE EXIT HANDLER FOR user-defined-condition-name
BEGIN

 ...statements...

END;
```

Take note that both the `DECLARE CONDITION` and `DECLARE HANDLER` Statement should be placed in the `BEGIN` section and not in the `DECLARE` section of the code.

Finally, use the `SIGNAL` Statement to cause a user-defined condition to be executed. Note that for both `SQLEXCEPTION` and `NOT FOUND`, using the `SIGNAL` Statement is not necessary.

For example, let's alter our divide by zero code to use a user-defined condition called `zero_divide`:

```
set hplsql.onerror=exeception;
DECLARE

 dVAL double := -999.0;
 dNUM int := 5;
 dDEN int := 0;

BEGIN

 --DECLARE A USER-DEFINED CONDITION BELOW.
 DECLARE zero_divide CONDITION;
```

```
 DECLARE EXIT HANDLER FOR zero_divide
 BEGIN
  PRINT ">>>>> DIVISION BY ZERO IS A NO-NO!! <<<<<";
 END;

 --DIVIDE dNUM BY dDEN...WHAT COULD POSSIBLY GO WORNG?
 IF dDEN = 0 THEN

  SIGNAL zero_divide;

 ELSE

  dVAL := dNUM / dDEN;

 END IF;

END;
```

As you see, the code checks if the denominator is zero (`dDEN = 0`) and, if so, the user-defined condition `zero_divide` is raised using the `SIGNAL` Statement. The output from this code is as follows:

```
>>>>> DIVISION BY ZERO IS A NO-NO!! <<<<<
```

Capturing SQL Errors

As indicated above, both `SQLEXCEPTION` and `NOT FOUND` don't require the use of the `SIGNAL` Statement, but will automatically be raised if a SQL error occurs. Note that when using these two handler options, there's no need to use `DECLARE CONDITION`. For example, here's how to declare an `EXIT` handler for `SQLEXCEPTION`:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
BEGIN

 ...statements...

END;
```

For example, let's write some garbage SQL and try to capture the error:

```
set hplsql.conn.default=impala;
set hplsql.onerror=exeception;
DECLARE

 iCNT int := -1;
 sSQL string;

BEGIN

 DECLARE EXIT HANDLER FOR SQLEXCEPTION
 BEGIN
  PRINT ">>>>> UH-OH! THERE'S BEEN A SQL ERROR!! <<<<<";
 END;

 sSQL := "
          SELECT COUNT(DISTINCT STATE_CODE) AS DIST_STATE
           FROM DIM_POSTAL_CODA
         ";
 EXECUTE sSQL INTO iCNT;
 PRINT iCNT;
```

```
END;
```

Take note that the table DIM_POSTAL_CODE has been misspelled as DIM_POSTAL_CODA. Utterly shameful!! Here's the output:

```
>>>>> UH-OH! THERE'S BEEN A SQL ERROR!! <<<<<
```

It's nice to create your own error message for a SQL exception, but if you'd like to retrieve the actual SQL error message, you can use the GET DIAGNOSTICS Statement:

```
GET DIAGNOSTICS EXCEPTION 1 hplsql-variable-name = MESSAGE_TEXT;
```

Let's change the DECLARE HANDLER above to grab the error message:

```
DECLARE EXIT HANDLER FOR SQLEXCEPTION
 BEGIN
  GET DIAGNOSTICS EXCEPTION 1 sSQLERR = MESSAGE_TEXT;
  PRINT ">>>>> UH-OH! THERE'S BEEN A SQL ERROR!! <<<<<";
  PRINT "THE ACTUAL ERROR MESSAGE IS => " + sSQLERR;
 END;
```

Now, when this program is executed, the following messages are printed out:

```
>>>>> UH-OH! THERE'S BEEN A SQL ERROR!! <<<<<
THE ACTUAL ERROR MESSAGE IS => [Cloudera][ImpalaJDBCDriver](500051) ERROR
processing query/statement. Error Code: 0, SQL state:
TStatus(statusCode:ERROR_STATUS, sqlState:HY000,
errorMessage:AuthorizationException: User 'smithbob' does not have privileges
to execute 'SELECT' on: prod_schema.dim_postal_coda
), Query: SELECT COUNT(DISTINCT STATE_CODE) AS DIST_STATE FROM DIM_POSTAL_CODA.
```

Excellent!!

Now, if you're creating a cursor, you can declare a NOT FOUND handler to capture any error that occurs due to incorrect SQL being passed to the cursor:

```
set hplsql.conn.default=impala;
set hplsql.onerror=exeception;
DECLARE

 iCNT int := -1;
 sSQL string;

BEGIN

 DECLARE EXIT HANDLER FOR NOT FOUND
 BEGIN
  PRINT ">>>>> NOT FOUND! <<<<<";
 END;

 OPEN csrSTATECODES FOR 'SELECT DISTINCT STATE_CODE FROM DIM_POSTAL_CODA";

END;
```

Here's the output:

```
>>>>> NOT FOUND! <<<<<
```

PART VI - Updating Your Database

Chapter 29 – Database Import/Export Using sqoop

In *Chapter 1 – Quick Start Guide*, we pulled data from a remote database using `sqoop`, a command line utility which allows you to *import data from* or *export data to* a remote database. Although we placed the data in a temporary table, we then quickly followed up by issuing an ImpalaSQL query to create a final table with the correct data types and storage format.

In this chapter, we discuss how to import tables from a remote database directly into Hadoop as well as export Hadoop tables to a remote database. We assume that the remote database is your legacy database, but that's not necessary and can be any database. Note that `sqoop` makes use of Java `.jar` files to communicate with the remote database. For example, the Java `.jar` file I'll be using to connect to a remote MySQL database is named `mysql-connector-java-#.#.#.jar`. Your death-defying Hadoop Administrator should have the name and location of the appropriate Java `.jar` file used to connect to your legacy database and it should be located in the appropriate directory where `sqoop` can pick it up automagically.

Note that, for each remote database you want to interact with using `sqoop`, there's a corresponding JDBC connection string required to start the conversation. Recall in *Chapter 2 – Hadoop Administrator E-Mail*, we asked for the ability to access the legacy database from the Linux edge node server. Before attempting to use `sqoop` with your legacy (or other) database, ensure that you can connect to it using database-specific software, if possible. For example, if the remote database is a MySQL database, you can use the Linux command line utility `mysql` to log into the remote database (assuming the administrator has granted you permission). If the remote database is Oracle, you can use the SQL*Plus command `sqlplus` (with an appropriately updated `tnsnames.ora` file) to log into the database, or you can use `tnsping` to ping the remote database. If you're having trouble connecting to the remote database at this point, you may need to involve more than just your Hadoop Administrator (e.g., Linux Administrator, Database Administrator, Network Administrator, Party Administrator, etc.) in order to resolve the connection problem before using `sqoop`.

Simple Example

At its very simplest, `sqoop` allows you to list the tables in the remote database schema you're connecting to. For example, let's display a list of tables in the remote MySQL database schema `retail_db`:

```
sqoop list-tables
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
```

Note that you'll have to update the connection string above with the correct remote database host and schema as well as enter in the correct *username* and *password*, of course.

Here's the output I see when using the MySQL database in the Cloudera QuickStart virtual machine we discussed in *Chapter 5 – Creating Your Very Own Hadoop Playground*:

```
INFO sqoop.Sqoop: Running Sqoop version: 1.4.7-cdh6.3.2
WARN tool.BaseSqoopTool: Setting your password on the command-line is insecure.
Consider using -P instead.
INFO manager.MySQLManager: Preparing to use a MySQL streaming resultset.
categories
customers
departments
order_items
orders
products
[smithbob@lnxserver ~]$
```

This is a very nice, calm, relaxing way to test `sqoop` against the remote database! If you'd like to list the database schemas instead then replace `list-tables` with `list-databases`:

```
sqoop list-databases
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
```

And here are the results (without all of those eye-watering messages):

```
information_schema
amon
metastore
mysql
oozie
performance_schema
retail_db
rman
scm
[smithbob@lnxserver ~]$
```

`sqoop list-databases`	Displays a list of databases/schemas in the remote database.
`sqoop list-tables`	Displays a list of tables in the remote schema.

Can We Talk?

There are three main ways in which `sqoop` interacts with a remote database:

- ☐ `import` (HDFS **without** Hive support) – This option pulls data from a table located in a remote database and places it directly in HDFS. The Hive MetaStore won't be informed that the table exists, though, and you'll have to use the `CREATE EXTERNAL TABLE` Statement in order to access the table's data.
- ☐ `import` (HDFS **with** Hive support) – Similar to the option above, but places the table's metadata information in the MetaStore so that Hive immediately recognizes it as a table. In order for Impala to recognize the table, you must issue `INVALIDATE METADATA` *`table-name`*`;` on the table after the `sqoop` command completes by using, say, the Linux command line utility `impala-shell` or your SQL Client.
- ☐ `export` – This option allows you to export a table's data from the Hadoop database into a table on the remote database. Note that the table must already exist in the remote database, so a `CREATE TABLE` Statement must be issued prior to performing an export using `sqoop`.

`sqoop import...`	Imports a table from the remote database into Hadoop.
`sqoop export...`	Exports a table from Hadoop to the remote database.

We discuss the import options related to the Hive MetaStore later in this chapter.

When importing data, you can specify which storage format you'd like `sqoop` to create using one of the following switches on the `sqoop` command line:

- ☐ `--as-parquetfile` – This option tells `sqoop` to store the table in the Parquet format.
- ☐ `--as-textfile` – This option tells `sqoop` to store the table in the Textfile format.

The default is `--as-textfile`. Please see the documentation for more storage formats.

`sqoop import... --as-parquetfile`	Imports the table in the Parquet storage format.
`sqoop import... --as-textfile`	Imports the table in the Textfile storage format.

If you'd like the data file(s) to be compressed, you can specify the compression switch as well as the desired compression codec:

- `--compress` – Turns on compression with `gzip` as the default compression codec.
- `--compress --compression-codec opt` – Turn on compression with a specific compression codec. But, please re-read the section entitled **Saving Space with COMPRESSION_CODEC** in *Chapter 16 – SQL Performance Improvements* when deciding on a compression codec.

`sqoop import... --compress`	Compresses the imported table using the `gzip` compression codec.
`sqoop import... --compress --compression-codec opt`	Compresses the imported table using the specified compression codec *opt*.

Specifically for the `import` option, I like to think of the `sqoop` command line utility itself as being broken up into several sections each with its own specific options:

```
sqoop import  Memory-related options
              Hive MetaStore-related options
              JDBC Connection options
              Performance Improvement options
              Storage options
              Target Directory options
              Data Type Fixing options
```

We discuss each of these below along with their specific options.

Finally, when running the `sqoop` utility, you may see one or more Java programs magically appearing in the Linux directory where you executed `sqoop`. These are just Java files created by `sqoop` to import or export your data and can be deleted after `sqoop` has completed.

Data Type Conversion Issues

As much as we'd all love to think of the world as a beautiful place full of unicorns and fluffy bunnies, occasionally a unicorn sneezes and impales a fluffy bunny dead. **sniff sniff** The same can be said with data type conversion from the remote database to the Hadoop database. For the most part, `sqoop` does an excellent job especially if the data types available in the remote database are similar to the data types available in Hive/Impala. For example, the data types of `BIG INT`, `INT`, `SMALL INT`, `TINY INT`, `STRING`, etc. seem to import with almost no issues. Data types related to dates and times can be problematic and may need to be finessed after being imported. And, data types such as character large objects (`CLOB`s), binary large objects (`BLOB`s), etc. may require additional `sqoop` options. Many of these issues stem from non-standard data types such as Oracle's `NUMBER` or `CLOB` data types. There may be additional issues with your remote (legacy) database whether it be Oracle, Microsoft SQL Server, Teradata, etc. you may need to work around and we suggest a few workarounds below.

One simplistic workaround is to create a SQL query to be executed on the remote database (using `sqoop`'s `--query` switch) which just converts some of these problematic data types into `STRING`s. Once `sqoop` is finished importing the table's data, you can use the `CAST()` function to transmogrify the `STRING` into something you really want…like a living fluffy bunny…or even a `TIMESTAMP`. For example, let's convert the column `purchase_date` in the Oracle table `purchase` to a `STRING` in `YYYYMMDD` format using Oracle's `TO_CHAR()` function (connection information left off and code formatted for OCD clarity):

```
sqoop import
     --query "select purchase_id,
                     to_char(purchase_date,'YYYYMMDD') as purchase_date,
                     purchase_item
                from purchase
                where \$CONDITIONS"
```

Rather than issuing a SQL query on the remote database, you can specify a table name with the `--table` switch, but the conversion above, naturally, won't happen:

```
sqoop import
      --table purchase
```

`sqoop import... --query "...sql..."`	Execute *sql* on the remote database and retrieve the resultset.
`sqoop import... --table table-name`	Just import the data contained in the table *table-name*.

Another way to help the conversion issue is to use `--map-column-hive` or `--map-column-java`, two `sqoop` switches which indicate your desired target data type. For example, to tell `sqoop` that you want the Oracle `CLOB` column `unnecessarily_verbose_description` to be converted to a `STRING` data type, you can specify something like the following:

```
sqoop import
      --table purchase
      --map-column-java unnecessarily_verbose_description=STRING
```

`sqoop import... --map-column-hive col1=dt1,...`	Map column *col1* to data type *dt1*, etc. for Hive.
`sqoop import... --map-column-java col1=dt1,...`	Map column *col1* to data type *dt1*, etc. for Java.

Dates and times imported from the remote database can be problematic as they may be converted by `sqoop` to `BIG INT`s containing the number of milliseconds since the Unix Epoch. To convert such a beast to a `TIMESTAMP`, say, you can use something similar to the code below:

```
CAST(INPUT_DATE/1000 AS TIMESTAMP) AS FINAL_DATE
```

Naturally, this will vary depending on your database, so please check the results against the remote (legacy) database.

Memory-Related Options

Occasionally, an import will fail with a memory-related error message. You can increase available memory using a combination of the following four options following the `-D` switch on the `sqoop` command line:

- `mapreduce.`**`map`**`.memory.mb=#` – This option allows you to increase the memory provided to the mappers up to the specified value.
- `mapreduce.`**`reduce`**`.memory.mb=#` – This option allows you to increase the memory provided to the reducers up to the specified.
- `mapreduce.`**`map`**`.java.opts=-Xmx#g` – This option allows you to increase the memory provided to the Java mappers up to the specified value.
- `mapreduce.`**`reduce`**`.java.opts=-Xmx#g` – This option allows you to increase the memory provided to the Java reducers up to the specified.

For me, the two options `mapreduce.`**`map`**`.memory.mb` and `mapreduce.`**`map`**`.java.opts` seem to correct any memory-related issues related to importing from a remote database. It's recommended that the value you specify for `mapreduce.`***`X`***`.java.opts` be less than the value of the corresponding `mapreduce.`***`X`***`.memory.mb` option, but I've gotten away with both values being the same and the authorities haven't taken me away yet.

You specify these options on the `sqoop` command line like this:

```
sqoop import
      -Dmapreduce.map.memory.mb=32768
      -Dmapreduce.map.java.opts=-Xmx16g
      --table purchase
```

Please examine the amount of available RAM on your Linux edge node server and choose appropriate values that won't ignite the server on fire. And, as always, please contact your diamond-studded Hadoop Administrator if your imports are failing due to memory issues…or you're just lonely.

Import (HDFS without Hive Support)

In this section, let's pull some data from our remote MySQL database into HDFS using `sqoop`. In the example below, we pass in the following query to the remote database (normally, the SQL code would appear on one line, but my OCD has been triggered again…):

```
select category_id,category_department_id,category_name
 from retail_db.categories
 where \$CONDITIONS
```

Note that I'm requesting the columns `category_id`, `category_department_id` and `category_name` from the table `categories` located in the `retail_db` schema. The `WHERE` Clause shown above is a requirement of `sqoop` when used with `--query` and is used internally. You can subset the data by adding the appropriate conditions to the `WHERE` Clause, but tack `AND \$CONDITIONS` to the end of the query. The backslash (`\`) before the dollar sign (`$`) is used to escape the dollar sign to prevent it from being misinterpreted. Here's the full `sqoop` command (shown on multiple lines):

```
sqoop import
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
      --query "select category_id,category_department_id,category_name from
               retail_db.categories WHERE \$CONDITIONS"
      --target-dir /user/hive/warehouse/tmp_categories
```

Several incomprehensible lines of output will fly across your screen, but at the end you'll see the following summary (line-by-line timestamps have been removed for clarity):

```
INFO mapreduce.Job:  map 0% reduce 0%
INFO mapreduce.Job:  map 100% reduce 0%
INFO mapreduce.Job: Job job_1652113797137_0001 completed successfully
INFO mapreduce.Job: Counters: 33
       File System Counters
              FILE: Number of bytes read=0
              FILE: Number of bytes written=242714
              FILE: Number of read operations=0
              FILE: Number of large read operations=0
              FILE: Number of write operations=0
              HDFS: Number of bytes read=85
              HDFS: Number of bytes written=1029
              HDFS: Number of read operations=6
              HDFS: Number of large read operations=0
              HDFS: Number of write operations=2
              HDFS: Number of bytes read erasure-coded=0
...snip...
       File Input Format Counters
              Bytes Read=0
       File Output Format Counters
              Bytes Written=1029
INFO mapreduce.ImportJobBase: Transferred 1.0049 KB in 37.9923 seconds (27.0844 bytes/sec)
INFO mapreduce.ImportJobBase: Retrieved 58 records.
```

As you see above, `58` records have been retrieved from the query submitted to the remote database and placed in HDFS under the directory named `tmp_categories`. When we look at the corresponding HDFS directory, we see, in part, the following:

```
[smithbob@lnxserver ~]$ hadoop fs -ls -R /user/hive/warehouse/tmp_categories
-rw-r--r--   3 osboxes hive       1029 2022-05-09 23:04
                               /user/hive/warehouse/tmp_categories/part-m-00000
[smithbob@lnxserver ~]$
```

If we `cat` the file `part-m-00000` to the screen, this is what we'll see:

```
[smithbob@lnxserver ~]$ hadoop fs -cat
                               /user/hive/warehouse/tmp_categories/part-m-00000
1,2,Football
2,2,Soccer
...snip...
57,8,MLB Players
58,8,NFL Players
```

Since we didn't specify the Hive-related options (discussed in the next section), this table won't appear in Hive. To fix this situation, we can create an external table pointing to the location `/user/hive/warehouse/tmp_categories` (or your specific directory). Please see *Chapter 8 – The One About ImpalaSQL* and *Chapter 23 – Working with Managed and External Tables* for more on creating external tables.

Note that the `sqoop` switches `--connect`, `--username`, `--password`, and `--query` specify how to connect to the remote database as well as specify the query to be submitted directly to the remote database. The switch `--target-dir` allows you to specify exactly where in HDFS you want the table's data to be stored. This option gives you ultimate control as to where the data is located in HDFS.

I tend to stick with `--target-dir` because I can control the exact name of the directory in HDFS. Alternatively, you can specify the `--warehouse-dir` switch to indicate the **parent directory** and `sqoop` will name the table-specific directory itself based on the name of the table provided with the `--table` switch. For example, let's pull the entire table `categories` into HDFS:

```
sqoop import
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
      --table categories
      --warehouse-dir /user/hive/warehouse
```

Now, looking in the directory `categories` we see the following:

```
[smithbob@lnxserver ~]$ hadoop fs -ls -R /user/hive/warehouse/categories
-rw-r--r--   3 osboxes hive        271 2022-05-10 19:33
                                 /user/hive/warehouse/categories/part-m-00000
-rw-r--r--   3 osboxes hive        263 2022-05-10 19:33
                                 /user/hive/warehouse/categories/part-m-00001
-rw-r--r--   3 osboxes hive        266 2022-05-10 19:33
                                 /user/hive/warehouse/categories/part-m-00002
-rw-r--r--   3 osboxes hive        229 2022-05-10 19:33
                                 /user/hive/warehouse/categories/part-m-00003
[smithbob@lnxserver ~]$
```

In this case, `sqoop` produced four separate text files (*insert ominous music here!*), but since we'll specify the HDFS directory on the `LOCATION` Clause of the `CREATE EXTERNAL TABLE` Statement, ImpalaSQL will read all of the files in that directory as if it were one big honkin' file.

`sqoop import --connect "`*`conn_string`*`"`	Specifies how to connect to the remote database.
`sqoop import --username `*`username`*	Specifies the remote database username *`username`*.
`sqoop import --password `*`password`*	Specifies the remote database password *`password`*.

`sqoop import --target-dir` *`directory`*	Specifies the HDFS target directory *`directory`*.
`sqoop import --warehouse-dir` *`directory`*	Specifies the HDFS parent directory *`directory`*. Used with the `--table` switch.

Now, when using the `--table` switch, you can specify a comma-delimited list of desired columns for `sqoop` to pull down using the `--columns` switch. For example,

```
sqoop import
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
      --table categories
      --columns category_id,category_department_id,category_name
      --warehouse-dir /user/hive/warehouse
```

`sqoop import --table` *`table-name`*	Specifies the table *`table-name`* to import from the remote database.
`sqoop import --table` *`table-name`* `--columns` *`col1,col2,...`*	Specifies the desired columns *`col1`*, *`col2`*, etc. to import from the table *`table-name`*.

If you'd like to completely remove the directory before re-pulling the data, you can use the `--delete-target-dir` switch. For example, if the `sqoop` import code needs to be rerun, tell `sqoop` to first delete the directory and its contents by adding the `--delete-target-dir` switch to the `sqoop` command line:

```
sqoop import
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
      --table categories
      --columns category_id,category_department_id,category_name
      --warehouse-dir /user/hive/warehouse
      --delete-target-dir
```

If you'd like to append data to an existing table, you can use the `--append` switch:

```
sqoop import
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
      --table categories
      --columns category_id,category_department_id,category_name
      --warehouse-dir /user/hive/warehouse
      --append
```

Looking at the directory `/user/hive/warehouse/categories` now, we see several additional files:

```
[smithbob@lnxserver ~]$ hadoop fs -ls -R /user/hive/warehouse/categories
-rw-r--r--   3 osboxes hive              271 2022-05-10 19:33
                                   /user/hive/warehouse/categories/part-m-00000
-rw-r--r--   3 osboxes hive              263 2022-05-10 19:33
                                   /user/hive/warehouse/categories/part-m-00001
-rw-r--r--   3 osboxes hive              266 2022-05-10 19:33
                                   /user/hive/warehouse/categories/part-m-00002
-rw-r--r--   3 osboxes hive              229 2022-05-10 19:33
                                   /user/hive/warehouse/categories/part-m-00003
-rw-r--r--   3 osboxes supergroup        271 2022-05-10 19:53
                                   /user/hive/warehouse/categories/part-m-00004
-rw-r--r--   3 osboxes supergroup        263 2022-05-10 19:53
                                   /user/hive/warehouse/categories/part-m-00005
```

```
-rw-r--r--   3 osboxes supergroup        266 2022-05-10 19:53
                                   /user/hive/warehouse/categories/part-m-00006
-rw-r--r--   3 osboxes supergroup        229 2022-05-10 19:53
                                   /user/hive/warehouse/categories/part-m-00007
```

`sqoop import... --append`	Appends data from the remote database to an existing table.
`sqoop import... --delete-target-dir`	Deletes the directory specified by `--target-dir` or created from `--warehouse-dir`.

Import (HDFS with Hive Support)

In the previous section, we pulled table data from the remote database into a directory in HDFS. If that were the only available feature of `sqoop`, we'd all still be tremendously impressed. And as they say on those late night television commercials: *BUT WAIT...THERE'S MORE!* By specifying a few additional switches, `sqoop` notifies the Hive MetaStore as to the existence of the table along with its columns and data types…all without the need to code that damnable `CREATE EXTERNAL TABLE` Statement! Sure, it's not a set of ginsu steak knives, but still, that's pretty damn good! Now, along with all the shizz presented in the previous section, you can specify the following options as well:

- ☐ `--hive-database` *`schema`* – Specifies the schema in which the table should be placed.
- ☐ `--hive-table` *`table-name`* – Specifies the name of the table as Hive sees it.
- ☐ `--hive-import` – Imports the table into Hive.
- ☐ `--hive-overwrite` – Overwrites the existing data in the Hive table, if it exists.

In the examples shown in the previous section, the table `categories` never appears in Hive…really, I checked. Let's modify the code to force Hive to acknowledge the existence of our ginsu-quality table:

```
sqoop import
      --hive-database prod_schema
      --hive-table categories
      --hive-import
      --hive-overwrite
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
      --table categories
      --columns category_id,category_department_id,category_name
      --warehouse-dir /user/hive/warehouse
      --delete-target-dir
```

`sqoop import... --hive-database` *`schema`*	Add the table to the Hive MetaStore in the specified *`schema`*.
`sqoop import... --hive-table` *`table-name`*	Add the table to the Hive MetaStore as the table *`table-name`*.
`sqoop import... --hive-import`	Import the table into Hive.
`sqoop import... --hive-overwrite`	Overwrite the existing data in the table.

Note that you must be careful with your choice of `--target-dir` or `--warehouse-dir`. If the location of the final **managed** table is in the **same exact location**, this variation of the `sqoop` command may delete the files despite being successfully executed. The table may exist, but the underlying files will be missing and the table will be empty. So, ensure that you specify an HDFS directory in `--target-dir`/`--warehouse-dir` **away from where your managed tables are normally located**. In computer science, this is called a *yikes*!

You can view the table using Hive's `beeline` command line utility, but let's go into `impala-shell` and issue `INVALIDATE METADATA` on our table:

```
[smithbob@lnxserver ~]$ impala-shell
[hdpserver:21000] prod_schema> invalidate metadata categories;
[hdpserver:21000] prod_schema> select * from categories;
```

```
+-------------+------------------------+----------------------+
| category_id | category_department_id | category_name        |
+-------------+------------------------+----------------------+
| 1           | 2                      | Football             |
| 2           | 2                      | Soccer               |
...snip...
| 28          | 5                      | Top Brands           |
| 29          | 5                      | Shop By Sport        |
+-------------+------------------------+----------------------+
[hdpserver:21000] prod_schema>
```

Nice!

Satan's Anus: Handling `NULL`s

If your table contains any of those damnable `NULL` values, either for numeric or character columns, you can specify two additional `sqoop` parameters:

`sqoop ... --null-string 'value'`	Replace `NULL` with *`value`* in string columns.
`sqoop ... --null-non-string 'value'`	Replace `NULL` with *`value`* in non-string columns.

By default, the value ~~haggis~~ `null` is used if not specified with one or both of the parameters above. For example, in the code below, both parameters are set to double tickmarks (`''`) indicating `NULL` values will be replaced with an empty field indicating a `NULL`:

```
sqoop import
      --hive-database prod_schema
      --hive-table categories
      --hive-import
      --hive-overwrite
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
      --table categories
      --columns category_id,category_department_id,category_name
      --null-string ''
      --null-non-string ''
      --warehouse-dir /user/hive/warehouse
      --delete-target-dir
```

Hang On a Minute, Cheeky Monkey!!

When we queried the table `categories` using the `--query` switch, we got back a single data file: `part-m-00000`. And, when we pulled the table using the `--table` switch, we got back four data files: `part-m-00000`, `part-m-00001`, `part-m-00002`, and `part-m-00003`. What kinda mysterious magicwand-flinging-spell-casting crap is that?!? If you look at the definition of the table `retail_db.categories` **in the MySQL database**, you'll see that the column `category_id` is set as the primary key of that table, indicated below by `PRI` under the column `Key`:

```
mysql> desc categories;
+------------------------+-------------+------+-----+---------+----------------+
| Field                  | Type        | Null | Key | Default | Extra          |
+------------------------+-------------+------+-----+---------+----------------+
| category_id            | int(11)     | NO   | PRI | NULL    | auto_increment |
| category_department_id | int(11)     | NO   |     | NULL    |                |
| category_name          | varchar(45) | NO   |     | NULL    |                |
+------------------------+-------------+------+-----+---------+----------------+
```

In order for sqoop to speed up the transfer of data from the remote database into HDFS, sqoop automatically parallelizes the download based on a range of values from the primary key column(s). Naturally, a SQL query sent to the remote database using the --query switch doesn't have a primary key. When we used --table, sqoop parallelized the download using four processes (the default number), which is why there are four data files. You can see this in the log produced by sqoop:

```
INFO db.DataDrivenDBInputFormat: BoundingValsQuery:
                SELECT MIN(`category_id`), MAX(`category_id`)
                 FROM `categories`
INFO db.IntegerSplitter: Split size: 14; Num splits: 4 from: 1 to: 58
INFO mapreduce.JobSubmitter: number of splits:4
```

As you see, sqoop is determining the range of values of the primary key column category_id and then splitting them up into four chunks. Further below, in the log, you'll see the following summary indicating the number of launched tasks (4 in this case):

```
Job Counters
 Launched map tasks=4
 Other local map tasks=4
 Total time spent by all maps in occupied slots (ms)=12806656
 Total time spent by all reduces in occupied slots (ms)=0
 Total time spent by all map tasks (ms)=25013
 Total vcore-milliseconds taken by all map tasks=25013
 Total megabyte-milliseconds taken by all map tasks=12806656
```

When we used --query, sqoop couldn't parallelize the download, so only one process was executed and hence only one data file was created. But, there's a way to parallelize a SQL query passed in via the --query switch and we talk about that below.

In both cases (--query and --table), you can control the amount of parallelization you want, as we shall now discuss. If you're using the --query switch, you can specify an appropriate column to use to take advantage of parallelization. For example, let's tell sqoop to use the column category_department_id as the column to split on:

```
sqoop import
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
      --query "select category_id,category_department_id,category_name
               from categories
               where \$CONDITIONS"
      --split-by category_department_id
      --delete-target-dir
      --target-dir /user/osboxes/categories
      --hive-database default
      --hive-table categories
      --hive-import
      --hive-overwrite
```

In this case, four splits were used:

```
INFO db.DataDrivenDBInputFormat: BoundingValsQuery:
 SELECT MIN(category_department_id), MAX(category_department_id)
  FROM (
        select category_id,category_department_id,category_name
         from categories
         where  (1 = 1)
        ) AS t1
INFO db.IntegerSplitter: Split size: 1; Num splits: 4 from: 2 to: 8
```

```
INFO mapreduce.JobSubmitter: number of splits:4
```

But, you can control the number of splits – and, hence, the amount of parallelization – by using either the `-m` or `--num-mappers` switch (they're synonyms). For example, if `--num-mappers 2` is specified, two data files will be produced, not four:

```
sqoop import
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
      --query "select category_id,category_department_id,category_name
               from categories
               where \$CONDITIONS"
      --split-by category_department_id
      --num-mappers 2
      --delete-target-dir
      --target-dir /user/osboxes/categories
      --hive-database default
      --hive-table categories
      --hive-import
      --hive-overwrite
```

In the output, you'll see the following:

```
INFO db.DataDrivenDBInputFormat: BoundingValsQuery:
 SELECT MIN(category_department_id), MAX(category_department_id)
  FROM (
        select category_id,category_department_id,category_name
         from categories
         where  (1 = 1)
       ) AS t1
INFO db.IntegerSplitter: Split size: 3; Num splits: 2 from: 2 to: 8
INFO mapreduce.JobSubmitter: number of splits:2
```

And, in HDFS, you'll see only two data files now:

```
[smithbob@lnxserver ~]$ hadoop fs -ls -R
                  hdfs://quickstart-bigdata:8020/user/hive/warehouse/categories
-rwxrwxrwt   3 osboxes supergroup        403 2022-05-11 00:32
     hdfs://quickstart-bigdata:8020/user/hive/warehouse/categories/part-m-00000
-rwxrwxrwt   3 osboxes supergroup        626 2022-05-11 00:32
     hdfs://quickstart-bigdata:8020/user/hive/warehouse/categories/part-m-00001
[smithbob@lnxserver ~]$
```

Finally, if you'd like to serialize the download, just set the number of splits to `1`: `--num-mappers 1`.

Along with specifying the number of splits, you can tell `sqoop` how many rows you'd like to download in one chunk by using the `--fetch-size` switch. But, be aware that there's diminishing returns when using this switch and specifying the largest number you learned in college may not work to your advantage. For example, let's fetch a maximum of `10000` rows at a time:

```
sqoop import
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
      --query "select category_id,category_department_id,category_name
               from categories
               where \$CONDITIONS"
      --split-by category_department_id
```

```
--num-mappers 2
--fetch-size 10000
--delete-target-dir
--target-dir /user/osboxes/categories
--hive-database default
--hive-table categories
--hive-import
--hive-overwrite
```

`sqoop import... -m #`	Parallelize the download splitting the table into `#` pieces.
`sqoop import... --num-mappers #`	Synonym for the above switch.
`sqoop import... --split-by col1,col2,...`	If no primary key is available, or you'd like to override it, specify the column(s) to split on using this switch.
`sqoop import... --fetch-size rows`	Fetches *rows* rows at a time from the remote database.

Note that, if you're using the `--query` switch and attempting to pass in a voluminous SQL query, you can make your life easier by creating a view on the remote database and just simply selecting from the view with `--query`. Alternatively, you can just use `--table` on the view.

Now, when using `--table`, `sqoop` will use the primary key(s) when available, as we saw above, to slice-and-dice the remote database table into pieces to perform a parallelized download. Naturally, you can still specify either the `-m` or `--num-mappers` switch to control the number of parallel processes being executed.

The Oracle Speaks!

When attempting to import a very large table from Oracle using `sqoop`, you may run into an issue when using `--num-mappers` and `--split-by`, especially if your split-by column is highly skewed. This may cause a few of the mappers to do most of the work potentially slowing down the import. One way around this is to use the Oracle `MOD` function on your numeric split-by column to create an additional column named `SPLITCOL` (original, I know). It's this column which you'll name on the `--split-by` parameter. The number of mappers is specified both on the `--num-mappers` parameter as well as in the `MOD` function. For example, your query might look something like this:

```
SELECT *
 FROM (
       SELECT /*+ NO_PARALLEL */ A.*,
                                 MOD(A.YOUR_COLUMN,MAPPERS) AS SPLITCOL
        FROM YOUR_TABLE_NAME A
)
```

Note that you'll have to replace *`YOUR_COLUMN`* with the name of your numeric skewed column, *`MAPPERS`* with the number of mappers you're specifying on the `--num-mappers` parameter, and *`YOUR_TABLE_NAME`* with your table name. The `NO_PARALLEL` hint is used to prevent each `sqoop` mapper from itself being parallelized by the Oracle database. Naturally, you can control this by using Oracle's `PARALLEL` hint and specifying a **reasonable** number of threads/processes. In the code above, you might specify, say, `/*+ PARALLEL(A,2) */` to tell Oracle to use two threads/processes. Note that if you've specified `10` mappers and `2` threads, you'll have `20` threads/processes executing instead of only `10` when using the `NO_PARALLEL` hint. You may want to test several scenarios to determine which combination of values works best with your hardware.

Although you may be tempted to use Oracle's built-in `ROWNUM` pseudocolumn instead of *`YOUR_COLUMN`* to create `SPLITCOL`, you may want to avoid this because Oracle doesn't store tables sorted. This means that `ROWNUM` is created at query time and is not deterministic. You can prove this for yourself by querying a tiny portion of a large table several times noticing that a few different rows may be returned each time. This indeterminacy may cause duplicate rows to appear in the final table in Hadoop when `sqoop` completes. It's probably a good idea to ensure that your `SPLITCOL` is created in a deterministic manner; that is, each time the query is run, `SPLITCOL` returns the same value for each row in your large input table.

Export

In this section, we discuss how to export a table's data from Hadoop into a table on the remote database. For example, let's export the `candybar_consumption_data` from *Chapter 13 – Extensions to the GROUP BY Clause* into MySQL. Note that since MySQL uses the data type `VARCHAR` instead of `STRING`, we have to use the following code in MySQL when creating the table:

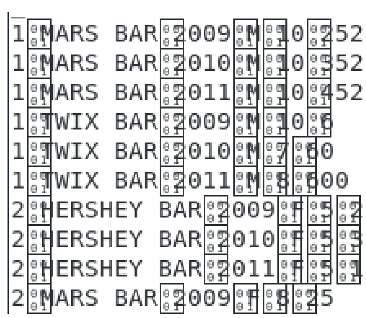
```
create table candybar_consumption_data(
 consumer_id tinyint,
 candybar_name varchar(20),
 survey_year smallint,
 gender varchar(1),
 overall_rating tinyint,
 number_bars_consumed smallint
);
```

Next, let's export the data from Hadoop into MySQL using the following `sqoop` command:

```
sqoop export
      --connect "jdbc:mysql://remotehost:3306/retail_db?serverTimezone=UTC"
      --username username
      --password password
      --table candybar_consumption_data
      --export-dir /user/hive/warehouse/candybar_consumption_data
      --fields-terminated-by '\001'
```

Similar to importing, we have to specify the connection string as well as the appropriate username and password. The table indicated on the `--table` switch is the name of the table in the **remote database**, NOT in Hadoop!! The `--export-dir` indicates the location in HDFS containing the data files for the table you want to export to the remote database. Finally, the switch `--fields-terminated-by` indicates how the fields are terminated within the data files themselves in HDFS. In this case, the default field terminator for the `TEXTFILE` storage format is `001`. If you `cat` the file itself, you'll see that the field terminator is a *wild thang*:

```
[smithbob@lnxserver ~]$ hadoop fs -cat
                  /user/hive/warehouse/candybar_consumption_data/blah_blah_data.0.
```



As usual, a blast of messages fly across the screen, but at the end you'll see a brief summary:

```
INFO mapreduce.ExportJobBase: Transferred 3.2119 KB in 33.7468 seconds
                                                          (97.4612 bytes/sec)
INFO mapreduce.ExportJobBase: Exported 36 records.
```

And, in MySQL, we can check out the table:

```
mysql> select * from candybar_consumption_data;
+-------------+---------------+-------------+--------+----------------+----------------------+
| consumer_id | candybar_name | survey_year | gender | overall_rating | number_bars_consumed |
+-------------+---------------+-------------+--------+----------------+----------------------+
|           5 | HERSHEY BAR   |        2010 | M      |              8 |                   15 |
```

```
|           5 | HERSHEY BAR   |        2011 | M      |              6 |                   5 |
|           5 | SNICKERS BAR  |        2009 | M      |              8 |                  55 |
|           5 | SNICKERS BAR  |        2010 | M      |              8 |                  65 |
|           5 | SNICKERS BAR  |        2011 | M      |              8 |                  75 |
...snip...
|           4 | MARS BAR      |        2011 | F      |              7 |                  15 |
|           4 | TWIX BAR      |        2009 | F      |              7 |                  20 |
|           4 | TWIX BAR      |        2010 | F      |              7 |                  30 |
|           4 | TWIX BAR      |        2011 | F      |              7 |                  10 |
|           5 | HERSHEY BAR   |        2009 | M      |              8 |                  15 |
+-------------+---------------+-------------+--------+----------------+---------------------+
36 rows in set (0.00 sec)
```

Success!! But, be aware that this form of `sqoop export` creates and executes one `INSERT` Statement for each row in the Hadoop data files. If there's data in the remote table already, you'll be effectively appending to it. This is fine if you're dropping and creating the table in the remote database as part of your process since the table will be empty. Or, you may be fine with appending data to the remote table. It all depends on you…so just be you! Note that `sqoop export` can perform `UPDATE` Statements rather than `INSERT` Statements. Please see the `sqoop` documentation for more on this option.

Party! Party! (Third) Party!

In this section, we discuss a few third-party tools which can be used to interact with your Hadoop database. We show no code in this section, rather we just spend some quality time discussing a few ways you can interact with Hadoop via third-party tools. In this case, though, importing/exporting is taking place on the legacy (remote) database side, not the Hadoop side. In the case of `sqoop`, importing/exporting is taking place on the Linux edge node server, not the remote database. Naturally, this section is in no way comprehensive…*ain't nobody got time fo' dat!*

The *TL;DR* for this section is simple: *Don't forget to have a discussion with your legacy database administrator as he/she may have one or more tools already installed on the legacy database which you can use to push data from the legacy database into the Hadoop database as well as pull data from Hadoop into the legacy database.*

As indicated earlier in the book, both ODBC and JDBC drivers can be used to make a connection between your software/database and the Hadoop database. For example, both Python and R have libraries/modules that can be installed which allow you to connect to Hadoop using the appropriate ODBC/JDBC drivers. As another example, both Tableau and Microsoft PowerBI make use of ODBC drivers to connect to Hadoop and pull data into their own internal storage or just pull directly from the database on an as-needed basis. We won't discuss ODBC and JDBC in this section, but be aware that there are also ODBC-JDBC/JDBC-ODBC bridge drivers available, if needed.

And, as we learned throughout this chapter, you can use `sqoop` to *import from* and *export to* your target database. In this case, though, the entity *leading the data mambo* isn't the database/application, but `sqoop` itself from the Hadoop side. In this section, we're more interested in the legacy (remote) database *leading the data mambo*. In other words, it's the legacy (remote) database side that will be doing the *pulling from* and *pushing to* Hadoop, not the Hadoop side.

We briefly discuss the following in the remainder of this section:

- ☐ Oracle – Oracle's Big Data Connectors suite of tools allow you to interact with Apache Hadoop.
- ☐ Microsoft SQL Server – SQL Server's PolyBase allows you to query a variety of databases, but there are other ways to connect to Hadoop from SQL Server.
- ☐ Teradata – Teradata's DataConnector Operator allows you to read Hadoop files and tables.
- ☐ SAS/ACCESS – SAS is a statistical analysis and data manipulation application and provides for access to a variety of databases via their SAS/ACCESS product suite.

Oracle Big Data Connectors

Oracle's Big Data Connectors, based on their marketing datasheet, is a suite of tools which integrates Apache Hadoop with your Oracle database allowing you to run SQL queries on vast amounts of data from within Oracle itself. The suite is comprised of the following components:

- Oracle Datasource for Apache Hadoop – This component allows you to access your Oracle database tables from Hadoop as if they were native Hadoop tables. SQL queries can issue joins between native Hadoop tables and Oracle tables. Results of a SQL query can be written directly to Oracle.
- Oracle SQL Connector for HDFS – This component allows you to query Hadoop directly from the Oracle database.
- Oracle Loader for Hadoop – This component allows you to load data from Hadoop into Oracle.
- Oracle R Advanced Analytics for Hadoop – This component allows you to run R code in Hadoop and Spark directly from Oracle.
- Oracle XQuery for Hadoop – This component allows you to use the power of XQuery against XML, JSON and other formats and the resulting data can be loaded into Oracle.
- Oracle Data Integrator – This graphical user interface (GUI) allows you to integrate Hadoop into your ODI projects.

You can find out more at https://www.oracle.com/database/technologies/datawarehouse-bigdata/big-data-connectors.html.

Microsoft SQL Server PolyBase

Microsoft SQL Server PolyBase allows you to use the built-in procedural language T-SQL to directly query external databases such as Oracle, Teradata, Hadoop, and a variety of other databases. Similar to Oracle Datasource for Apache Hadoop, you can join tables in Hadoop with tables in SQL Server. You can also query Hadoop tables from within SQL Server using data virtualization along with PolyBase connectors.

Unfortunately, support for both Cloudera Data Platform (CDP) and Hortonworks Data Platform (HDP) will be retired and not be included in Microsoft SQL Server 2022. You can find Microsoft's suggested replacement options here: https://docs.microsoft.com/en-us/sql/big-data-cluster/big-data-options?view=sql-server-ver15.

You can find out more about Microsoft SQL Server PolyBase at https://docs.microsoft.com/en-us/sql/relational-databases/polybase/polybase-guide?view=sql-server-ver15.

Teradata DataConnector Operator

The Teradata DataConnector Operator, along with the Teradata Connector for Hadoop (TDCH), provides access to read and write Hadoop files and tables.

You can find out more about the Teradata DataConnector Operator at https://docs.teradata.com/r/Teradata-Parallel-Transporter-Reference/July-2017/DataConnector-Operator as well as https://docs.teradata.com/r/Teradata-Parallel-Transporter-Reference/July-2017/DataConnector-Operator/Usage-Notes/Processing-Hadoop-Files-and-Tables.

SAS and Hadoop

If you're a user of SAS, you'll know that SAS can connect to a variety of databases using a SAS LIBNAME or PROC SQL with one of the many available SAS/ACCESS products. As with many SAS/ACCESS products, once connected to the Hadoop database, you can interact with the database tables as if they were SAS datasets as well as query the database using PROC SQL to send complex SQL queries directly to the Hadoop database. SAS has a variety of ways to interact specifically with a Hadoop database:

- SAS/ACCESS Interface to ODBC – This component allows you to access Hadoop using an appropriate ODBC driver.

- ☐ SAS/ACCESS Interface to Hadoop – This component allows you to query Hadoop and HDFS directly from SAS.
- ☐ SAS/ACCESS Interface to Impala – This component provides direct access to Impala from SAS.
- ☐ SQOOP Procedure – The SQOOP procedure allows you to submit SQOOP commands to your Hadoop cluster from within a SAS session. This procedure requires the purchase of SAS/ACCESS to Hadoop.
- ☐ HADOOP Procedure – The HADOOP procedure allows you to submit `hadoop`/`hdfs` commands, MapReduce programs and Pig Latin code to the cluster.
- ☐ SAS Data Loader for Hadoop – This point-and-click web application allows you to move, clean and analyze data in Hadoop.

You can find more information about SAS and its integration with Hadoop at https://documentation.sas.com/doc/en/hadoopov/9.4/p1d3oooypq5aemn1e3t2cbkvxm6p.htm.

Chapter 30 – Loading Data using `LOAD DATA` to Load Data

If, as part of your process, text files are loaded directly into HDFS to be used with, say, a particular table, you can use the `CREATE EXTERNAL TABLE` Statement, as we've seen, to access the underlying files and query the table. But, you can also use ImpalaSQL's `LOAD DATA` Statement to do a similar thing. Be aware that, unlike `CREATE EXTERNAL TABLE`, the `LOAD DATA` Statement **moves your files** from their current external location in HDFS to a managed location in HDFS.

To be completely honest, I'm not a tremendous fan of the `LOAD DATA` Statement and much prefer using `CREATE EXTERNAL TABLE` with `INSERT INTO SELECT` to load data into my target table. But, hey, that's probably just me!

Here are the steps you can employ to use the `LOAD DATA` Statement:

1. Determine the external HDFS location of the file(s) for the table.
2. Using ImpalaSQL, create your table.
3. Using ImpalaSQL, use the `LOAD DATA` Statement to load the data into the table.

Now, a few nasty things may happen along this journey, so let's do one of my patented examples using the tab-delimited text file `dim_postal_code.tsv`. First, let's make a copy of the file and call it `new_postal_code.tsv`:

```
[smithbob@lnxserver ~]$ cp dim_postal_code.tsv new_postal_code.tsv
```

Next, let's create a directory under the external branch of HDFS to contain this file.

```
[smithbob@lnxserver ~]$ hadoop fs -mkdir
   hdfs://lnxserver.com:8020/warehouse/tablespace/external/hive/new_postal_code
```

And, let's copy the file `new_postal_code.tsv` to that directory:

```
[smithbob@lnxserver ~]$ hadoop fs -copyFromLocal
  /home/smithbob/new_postal_code.tsv
  hdfs://lnxserver.com:8020/warehouse/tablespace/external/hive/new_postal_code/
                                                              new_postal_code.tsv
```

Okay, let's check that the file made it there:

```
[smithbob@lnxserver ~]$ hadoop fs -ls -R
  hdfs://lnxserver.com:8020/warehouse/tablespace/external/hive/new_postal_code

-rw-r--r--   3 hdfs supergroup    1784376 2022-03-30 10:22
hdfs://lnxserver.com:8020/warehouse/tablespace/external/hive/new_postal_code/
                                                              new_postal_code.tsv
```

Next, let's ensure that the `LOAD DATA` Statement can read this directory by issuing a `chmod` on it:

```
[smithbob@lnxserver ~]$ hadoop fs -chmod 666
   hdfs://lnxserver.com:8020/warehouse/tablespace/external/hive/new_postal_code
```

Great! Now, in `impala-shell` or your SQL GUI Client, let's first create the table:

```
CREATE TABLE NEW_POSTAL_CODE(
 POSTAL_CODE  STRING,
 CITY         STRING,
 STATE_CODE   STRING,
 LATITUDE     DOUBLE,
 LONGITUDE    DOUBLE
)
STORED AS TEXTFILE
```

```
TBLPROPERTIES('transactional'='false');
```

Take note that we're making the table non-transactional because the `LOAD DATA` Statement has issues with transactional (ACID) tables.

Next, let's alter the table to indicate that the underlying data is tab-delimited:

```
ALTER TABLE NEW_POSTAL_CODE SET SERDEPROPERTIES('field.delim'='\t');
```

Next, let's use the `LOAD DATA` Statement to load our tab-delimited text file into the table:

```
LOAD DATA INPATH '/warehouse/tablespace/external/hive/new_postal_code'
 INTO TABLE NEW_POSTAL_CODE;
```

Let's see if that worked:

```
SELECT *
 FROM NEW_POSTAL_CODE
 LIMIT 10;

+-------------+-------------+------------+-----------+--------------------+
| postal_code | city        | state_code | latitude  | longitude          |
+-------------+-------------+------------+-----------+--------------------+
| 00623       | CABO ROJO   | PR         | 18.08643  | -67.15222          |
| 00633       | CAYEY       | PR         | 18.194527 | -66.18346699999999 |
| 00640       | COAMO       | PR         | 18.077197 | -66.359104         |
| 00676       | MOCA        | PR         | 18.37956  | -67.08423999999999 |
| 00728       | PONCE       | PR         | 18.013353 | -66.65218          |
| 00734       | PONCE       | PR         | 17.999499 | -66.643934         |
| 00735       | CEIBA       | PR         | 18.258444 | -65.65987          |
| 00748       | FAJARDO     | PR         | 18.326732 | -65.652484         |
| 00766       | VILLALBA    | PR         | 18.126023 | -66.48208          |
| 00771       | LAS PIEDRAS | PR         | 18.18744  | -65.87088          |
+-------------+-------------+------------+-----------+--------------------+
```

Perfect! Or, not so perfect…let's describe the table using the `FORMATTED` option:

```
DESC FORMATTED NEW_POSTAL_CODE;
```

Without displaying all of the gory details, the rows to note are **`Table Type`** and **`Location`**:

```
Location: hdfs://lnxserver.com:8020/warehouse/tablespace/external/hive/
                                                          new_postal_code
Table Type: EXTERNAL_TABLE
```

Notice that the table is marked as `EXTERNAL TABLE` and the location points to the external branch of HDFS. This is contrary to the `LOAD DATA` Statement's documentation. Note that this seems to occur due to the `transactional=false` option, but without it the `ALTER TABLE` to specify the delimiter doesn't work. On the other hand, we can mark the table as `MANAGED` by issuing the following `ALTER TABLE` Statement:

```
ALTER TABLE NEW_POSTAL_CODE SET TBLPROPERTIES('EXTERNAL'='FALSE');
```

Now, the table is marked as `MANAGED`, but the location doesn't change to the managed branch of HDFS!

Note that you can overwrite the data in the table by providing the `OVERWRITE` Clause:

```
LOAD DATA INPATH '/warehouse/tablespace/external/hive/new_postal_code'
 OVERWRITE INTO TABLE NEW_POSTAL_CODE;
```

If your table is partitioned, you can also control which partitition the `LOAD DATA` Statement loads data into by using the `PARTITION` Clause along with hard-coded values for the partition(s):

```
LOAD DATA INPATH '/directory/...'
 OVERWRITE INTO TABLE partitioned-table-name
 PARTITION (partition-column-1 = value-1,
            partition-column-2 = value-2,
            ...
            partition-column-n = value-n);
```

Don't forget to place quotes around *`value-i`* for textual columns. For example,

```
LOAD DATA INPATH '/directory/new_jersey/...'
 INTO TABLE USA_WEATHER_DATA
 PARTITION (state_code = 'NJ');
```

Be aware that HiveQL has a `LOAD DATA` Statement as well and includes the `LOCAL` keyword which allows you to load data directly from the Linux filesystem. Please see the HiveQL documentation for more.

Chapter 31 – Scheduling Jobs Using `crontab`

As indicated earlier in the book, once you have a Linux script written and tested, it's easy to schedule it to run one or more times using the job scheduler cron. You simply add the desired starting date/time information in a file called the crontab file accessible via the Linux command line utility `crontab`. This utility allows you to edit the crontab file using the good ol' `vi` Editor.

Although cron is very simple to use, be aware that your company's IT department may want you to use a hardy-artisinal-firm-of-buttocked piece of software to schedule jobs. Please check with your Linux Administrator to find out more. With that said, using cron is simple and a great starter…much like the *foie gras* at a fancy dinner party.

Editing the `crontab` File

To edit the crontab file using the `vi` Editor, you issue the following command at the Linux command prompt:

```
[smithbob@lnxserver ~]$ crontab -e
```

Initially, the crontab file will be blank and you'll just see the familiar run of tildes down the left side of the screen in typical `vi` stylie:

```
~
~
~
~
~
~
~
~
~
~
~
~
"/tmp/crontab.zEaA9w" 0L, 0C
```

Take note of the emboldened file name above. This is only a temporary file and is not something you want to edit directly, which is why the `crontab` utility offers the `-e` option allowing you to edit the crontab file. Although we discuss the layout of this file in detail in the next section, enter `vi` **i**nsert mode and type in the following:

```
0 0 1 JAN * /home/smithbob/dbupdate
```

This line indicates that the script `dbupdate`, located in the `/home/smithbob` directory, will run at midnight on January `1`st each year. Incredible!!

When you're satisfied with your entry, save and exit out of the `vi` Editor in the normal manner (`ESC` + `:wq`). Now, `crontab` will display the following:

```
crontab: installing new crontab
[smithbob@lnxserver ~]$
```

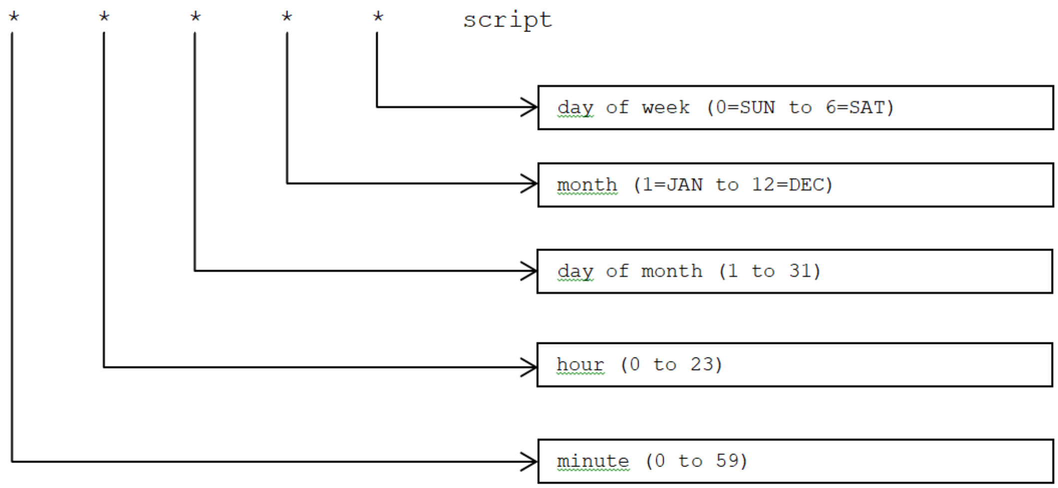
This indicates that your job has been scheduled to run. Woo-hoo! That was easy!!

If you'd like to dump the contents of the crontab file to the screen, use the `-l` switch instead of `-e`.

`crontab -e`	**e**dit	Edit the crontab file using the `vi` Editor.
`crontab -l`	**l**ist	Dumps the contents of the crontab file to the screen

Understanding the `crontab` Layout

Although the layout of the crontab file looks fairly cryptic, it really isn't. You start off with scheduling information followed by the location/name of the script you want scheduled to run. In general, this is what the format looks like:



The five fields can either contain an asterisk (`*`) or the indicated value (in parentheses above). Note that for day of week and month, you can substitute the three-letter day or month name in place of the numeric value, as indicated above.

If you set all five fields to asterisks (`*`), your script will be scheduled to run every minute. But, you can modify this by providing a *step value* in the form **`/#`**. For example, instead of every minute, let's run the script every five minutes:

```
*/5  *  *  *  *  /home/smithbob/dbupdate
```

Instead, let's schedule the script to run at `11:30` **AM** every day:

```
30  11  *  *  *  /home/smithbob/dbupdate
```

Note that the minutes appear to the left of the hour!

On second thought, let's run the script at `11:30` **PM** every day (note that cron uses a `24`-hour clock):

```
30  23  *  *  *  /home/smithbob/dbupdate
```

You know, we can probably get away with running the script the third day of every month at `2:15` AM:

```
15  02  3  *  *  /home/smithbob/dbupdate
```

Come to think of it, we can probably run the script once a year on the `15`th of October at `6:45` AM:

```
45  06  15  OCT  *  /home/smithbob/dbupdate
```

Oy! Sorry for the indecisivenessnessness, but let's run the script every Thursday at `1` AM:

```
00  01  *  *  THU  /home/smithbob/dbupdate
```

Okay, okay, I got it! Let's run the script the last day of each month at `3` AM. This is more difficult because the last day of every month can be a `28` (ignoring leap years here), `30` or `31` depending on the month. No problem! Just enter several lines in the crontab file along with a comma-delimited list of the desired numeric month values:

```
00  03  28  2                *  /home/smithbob/dbupdate
00  03  30  4,6,9,11         *  /home/smithbob/dbupdate
00  03  31  1,3,5,7,8,10,12  *  /home/smithbob/dbupdate
```

Again, we're using the numeric month values rather than the three-letter names.

Now, suppose you want to ensure that the script is run `1` AM on the **first** Friday of every month. Initially you'll try the following:

```
00  01  *  *  FRI  /home/smithbob/dbupdate
```

But, this will run the script **every** Friday at `1` AM, not the **first** Friday. In order to accomplish this, note that the first Friday of each month will have a day number less than or equal to `7`; that is, within the first seven days of the beginning of each month, there's bound to be a Friday. To accomplish this, we can use the Linux `date` utility to add in an additional check:

```
00  01  *  *  FRI  [ $(date +\%d) -le 07 ] && (/home/smithbob/dbupdate)
```

The command in square brackets checks if today's date's day is less than or equal to seven. If so, cron proceeds to run the script indicated in the parentheses. The two ampersands (`&&`) indicates a logical `AND` condition. If the day is greater than seven, the script is not executed. Unbelievable!!

You can always check your crontab fields by visiting the very useful website `https://crontab.guru`.

The crontab File and Kerberos keytab File

If your network is running the fab Kerberos computer-network authentication protocol, you'll have to do a little more work to get cron to run your jobs. From Wikipedia,

```
Kerberos is a computer network authentication protocol that works on the basis
of tickets to allow nodes communicating over a non-secure network to prove
their identity to one another in a secure manner.  Your mother wears army
boots.
```

[Wikipedia really shouldn't let me update wiki pages!]

Normally, when you log in to the Linux edge node server via PuTTY, you're proving who you are by providing both your username and password. When running cron jobs, it's a little more difficult since the job runs automatically and you're not there to enter in your password. Normally, this isn't a problem except when your network is running Kerberos or other network authentication protocol.

One way around this is to create a Kerberos *keytab file* which contains encrypted password information. This allows you to run cron jobs as long as you indicate where your Kerberos keytab file is located. Below are the instructions to create a Kerberos keytab file as well as how to use it within the crontab file.

To determine your username (or the generic account's username) as Kerberos sees it, type `klist` at the Linux command prompt:

```
klist
```

You should see something similar to the following:

```
Ticket cache: FILE:/tmp/krb5cc_123083789_0DLaGV
Default principal: SmithBob@COMPANY.COM

Valid starting       Expires              Service principal
01/29/2022 07:54:34  01/29/2022 17:54:34  krbtgt/COMPANY.COM@COMPANY.COM
        renew until 02/05/2022 07:54:34
```

Take note of the text following **Default principal:**. This will be needed later when scheduling jobs using cron. Now, at the Linux command prompt, type the following:

```
ktutil
```

This will start the Kerberos keytab file maintenance utility and the Linux command prompt will be replaced with the **ktutil:** prompt. Enter in the following making sure to change smithbob to either your username or the username of the generic account (which is a must if you're submitting jobs from that account!). After each line, you'll be prompted for the account password. Type in the password, then hit the Enter key to continue.

```
ktutil:  addent -password -p smithbob@lnxserver.company.com -k 1 -e rc4-hmac
Password for smithbob@lnxserver.company.com: <enter your password here>

ktutil:  addent -password -p smithbob@lnxserver.company.com -k 1 -e aes256-cts
Password for smithbob@lnxserver.company.com: <enter your password here>

ktutil:  addent -password -p smithbob@lnxserver.company.com -k 1 -e arcfour-hmac-md5
Password for smithbob@lnxserver.company.com: <enter your password here>

ktutil:  addent -password -p smithbob@lnxserver -k 1 -e rc4-hmac
Password for smithbob@lnxserver: <enter your password here>

ktutil:  addent -password -p smithbob@lnxserver -k 1 -e aes256-cts
Password for smithbob@lnxserver: <enter your password here>

ktutil:  addent -password -p smithbob@lnxserver -k 1 -e arcfour-hmac-md5
Password for smithbob@lnxserver: <enter your password here>

ktutil:  addent -password -p smithbob -k 1 -e rc4-hmac
Password for smithbob@COMPANY.COM: <enter your password here>

ktutil:  addent -password -p smithbob -k 1 -e aes256-cts
Password for smithbob@COMPANY.COM: <enter your password here>

ktutil:  addent -password -p smithbob -k 1 -e arcfour-hmac-md5
Password for smithbob@COMPANY.COM: <enter your password here>
```

Note that you're entering almost the same exact line of code several times with the exception of the encryption type (rc4-hmac, aes256-cts, arcfour-hmac-md5). Make sure to review the response to the Hadoop Administratior e-mail for information on which encryption types are necessary to include in the keytab file.

Finally, to save the Kerberos keytab file to disk, type the ktutil command wkt followed by the name of the output file (/home/smithbob/smithbob.keytab, say). To exit the app, type quit and hit the Enter key.

```
ktutil:  wkt /home/smithbob/smithbob.keytab
ktutil:  quit
```

Note #1: You may want to leave off aes256-cts if you receive the following error message during the kinit check (see further below):

```
kinit: Preauthentication failed while getting initial credentials
```

Note #2: If you still receive the error message above, you may want to uppercase lnxserver.company.com and add those lines to the keytab file as well. Please contact your Hadoop Administrator if you run into trouble.

Next, let's allow read access to our Kerberos keytab file:

```
chmod 644 /home/smithbob/smithbob.keytab
```

To test if the Kerberos keytab file is working, enter in the following at the Linux command prompt (making the necessary changes, of course):

```
kinit SmithBob@COMPANY.COM -k -t /home/smithbob/smithbob.keytab
```

If this command returns absolutely nothing, then you're good to go, buddy-boi!

Now that we have a Kerberos keytab file ready to go, here's how you use it with crontab. In the crontab editor (remember to issue `crontab -e` at the Linux command prompt), enter in the following (the example below runs the script every Tuesday at nightnoon) on one line:

```
0 0 * * TUE kinit SmithBob@COMPANY.COM -k -t /home/smithbob/smithbob.keytab;
/home/smithbob/dbupdate
```

The `-k` option tells crontab to use the Kerberos keytab file. The `-t` option indicates the location of the Kerberos keytab file. Take note of the semicolon!! Save and exit out of the crontab file and your job should run using cron without you being there! Huzzah!

Chapter 32 – Updating Your Hadoop Tables with `make`

Whether you're keeping your legacy database in the mix to serve as the master location for your fact and/or dimension tables, or you've gone rogue and cut over to Hadoop completely, you'll need to update your Hadoop database tables in `prod_schema` from time to time with new or updated data. As described in *Chapter 29 – Database Import/Export Using sqoop*, if pulling from the legacy database, you simply run `sqoop` to pull in the data and, maybe, run `impala-shell` to produce the final table. Once the data is all spiffified, you may also need to run a subsequent Python or R program to produce statistics, charts, additional tables, output files, mine cryptocurrency, etc. The world is your programmatic oyster!

With many tables, you can well imagine the number of `sqoop`, `beeline`, `impala-shell`, `python`, `RCMD`, etc. steps to execute each time an update is needed. Now, there are several ways to go here. You can create a text file containing all of the steps necessary to update each table and then copy-and-paste them one at a time to the Linux command prompt. This would not be my first port o' call.

Or, you can get fancy and create a Linux script containing a series of `case` Statements and pass in the name of the desired table to update as a parameter. Bah! I don't want to code all of those `case` Statements and not to mention the surrounding loop just in case you want to update multiple tables! Blech!! How about using Python? Nah…same problem as a Linux script and then you have to shunt out to the operating system using `os.system()` or other method. Oy! I don't have the strength! The pain, the pain!

There's a utility available in Linux called `make` used mainly to compile and link C and C++ programs to create executables. It's not necessarily made to do what I'm going to show you (*I await your tarring-and-feathering!*), but I've found that `make` is a quick, easy and cheap (read: *free!*) way to code for updates to the database. And no `if-then-else` statements required! Phew! Although many might think using `make` is overkill (or just plain wrong), I feel that the `make` utility along with its plain text control file, called a *makefile*, as described below, is a good compromise between functionality, maintainability and readability. After reading this chapter, you can decide whether it's a Linux script with all those dreadful `case` statements or a nice-easy-calming-luscious-fat-free *makefile* for you. No pressure.

The State of Baked Goods in the U.S.A.

Before we talk about `make`, let's assume that the table `DIM_US_STATE_MAPPING` in your legacy database has been updated due to some unhinged political correctness and `South Carolina` has been renamed `Pecan Pie Land`. Naturally, you'll use `sqoop` to import this table into Hive. After that, you'll run the corresponding SQL file `DIM_US_STATE_MAPPING.sql` in `impala-shell` (providing the fab `-f` switch, of course) to replace the table in Impala. This isn't some random, unhinged, lapses-of-the-s'napses comment since we'll actually use this information in the examples below.

Creating a `makefile` and Using `make`

The program `make` requires a driver file known as a *makefile* which contains rules for each table you want to update in the database. The file itself doesn't have to be called `makefile`, so let's call it `FabDeptDBUpdateFile`.

Now, if we're coding, say, a Linux script, we'd pass the table name as a parameter to the script and use an `if–then` statement, say, to find it and execute some code…

```
if [[ $1 == "DIM_US_STATE_MAPPING" ]] …
```

…or maybe a `case` Statement…

```
case $1 in
 "DIM_US_STATE_MAPPING")
   ...;;
esac
```

But, in the file `FabDeptDBUpdateFile`, we just have to place the following text starting in column `1`:

```
col 1
↓
DIM_US_STATE_MAPPING:
```

This is known as a *target* and you can provide several targets in `FabDeptDBUpdateFile`, one for each table you need to update. Take note that the target's name ends with a colon. Also, note that the target has a corresponding table name in the database (`PROD_SCHEMA.DIM_US_STATE_MAPPING`) as well as a corresponding SQL file (`DIM_US_STATE_MAPPING.sql`) that will be executed using `impala-shell`. We explain this more below, but be aware that you can arrange how this all works as you see fit.

The lines following the target name are where you code your `sqoop`, `impala-shell`, `python`, etc. executables to run, one command line per. For example (some parameters removed for clarity),

```
    col 1
    ↓
    DIM_US_STATE_MAPPING:
tab ⟶   echo "Target: $@"
tab ⟶   sqoop import --hive-database prod_schema ... --target-dir /data/prod/teams/prod_schema/TMP_$@
tab ⟶   impala-shell -k -i hdpserver --database prod_schema -f $@.sql
```

Note that `DIM_US_STATE_MAPPING` is the target name followed by a colon. The next line is just a simple `echo` command useful for debugging. The symbol **`$@`** automatically resolves to the target name (`DIM_US_STATE_MAPPING`, here). As you can see, the symbol **`$@`** is being used on the `sqoop` line to form the temporary target directory: `TMP_`**`$@`**. Here, this resolves to `TMP_`**`DIM_US_STATE_MAPPING`**. It's also being used on the `impala-shell` line where `-f` **`$@`**`.sql` resolves to `-f` **`DIM_US_STATE_MAPPING`**`.sql`.

One very important factoid: the lines following the target name MUST begin with a tab! Not 8 spaces, not the word `tab` in gold leaf, but a genuine, honest-to-goodness tab using the tab button on your keyboard!!

Now, you can create global variables near the top of `FabDeptDBUpdateFile` for use throughout the file. For example, let's create a global variable for our Hadoop schema `prod_schema`:

```
#------------------------------------------#
# Global variables                         #
#------------------------------------------#

# Target Impala database
TGTDB = prod_schema
```

We can then alter the code above to use the global variable `TGTDB` (whose variable resolution syntax below will look…hmmm!…vaguely familiar…):

```
    col 1
    ↓
    DIM_US_STATE_MAPPING:
tab ⟶   echo "Target: $@"
tab ⟶   sqoop import --hive-database ${TGTDB} ... --target-dir /data/prod/teams/prod_schema/TMP_$@
tab ⟶   impala-shell -k -i hdpserver --database ${TGTDB} -f $@.sql
```

Now, let's assume you've edited and saved the file `FabDeptDBUpdateFile` and are ready to take it for a spin to update South Carolina's new state name. From the Linux command line, issue the following command to update the table `DIM_US_STATE_MAPPING`:

```
make -f FabDeptDBUpdateFile DIM_US_STATE_MAPPING
```

Amazing!! The text you provide on the command line is just the **target name** you want to execute. The `make` utility will resolve any global variables and replace **`$@`** with the target name and then execute each tab-indented statement below it. Now, if you're a little skittish at first, you can have `make` display all of the steps **without actually executing any code** by providing the switch `--dry-run`:

```
make --dry-run -f FabDeptDBUpdateFile DIM_US_STATE_MAPPING

echo "Target: DIM_US_STATE_MAPPING"
sqoop import --hive-database prod_schema ... --target-dir
                                            /data/prod/teams/prod_schema/TMP_DIM_US_STATE_MAPPING
impala-shell -k -i hdpserver --database prod_schema -f DIM_US_STATE_MAPPING.sql
```

Okay, I can hear y'all saying, "I have to do that each time I want to update a single table?!?" Happily, you can provide multiple targets on the command line, like this:

```
make -f FabDeptDBUpdateFile DIM_US_STATE_MAPPING DIM_US_CENSUS_REGION
```

Assuming you're maintaining multiple tables – and I'm probably correct in this assumption – you can group together several targets into a *target group* and then enter the target group name on the command line instead. For example, let's create a *target group* named `usstates` which contains a space-delimited list of pre-existing *targets* in `FabDeptDBUpdateFile`:

```
usstates: DIM_US_STATE_MAPPING DIM_US_POSTAL_CODE DIM_US_CENSUS_REGION
```

Note that `FabDeptDBUpdateFile` looks like this now:

```
#------------------------------------------#
# Global variables                         #
#------------------------------------------#

# Target Impala database
TGTDB = prod_schema

# Target Groups
usstates: DIM_US_STATE_MAPPING DIM_US_POSTAL_CODE DIM_US_CENSUS_REGION

# Targets
DIM_US_STATE_MAPPING:
	echo "Target: $@"
	sqoop import --hive-database ${TGTDB} ... --target-dir /data/prod/teams/prod_schema/TMP_$@
	impala-shell -k -i hdpserver --database ${TGTDB} -f $@.sql

DIM_US_POSTAL_CODE:
	echo "Target: $@"
	sqoop import --hive-database ${TGTDB} ... --target-dir /data/prod/teams/prod_schema/TMP_$@
	impala-shell -k -i hdpserver --database ${TGTDB} -f $@.sql

DIM_US_CENSUS_REGION:
	echo "Target: $@"
	sqoop import --hive-database ${TGTDB} ... --target-dir /data/prod/teams/prod_schema/TMP_$@
	impala-shell -k -i hdpserver --database ${TGTDB} -f $@.sql
```

To update all three tables, just provide the **target group** rather than all three targets individually:

```
make -f FabDeptDBUpdateFile usstates
```

As you can well imagine, target groups can contain target groups as well as targets. So, mix it up a bit, kids!

Some Issues with $ and

Recall that you can pass a SQL query to `sqoop` after the `--query` or `-q` switch, but you must provide the text `\$CONDITIONS` in the `WHERE` Clause for `sqoop`'s own internal nefarious usage. Unfortunately, `make` attempts to resolve the `$C` and, not finding it, you wind up getting back `\ONDITIONS` instead of `\$CONDITIONS`. Horrendous! That type of thing really boils my butt! To solve this, though, double up on the dollar signs: `\$$CONDITIONS`. Now, `make` will return `\$CONDITIONS` and the world is a safer place once again.

Also, you may have some issues with the pound sign (`#`). To resolve this, escape each pound sign with a backslash: `\#` will resolve to `#`.

PART VII - Advanced Topics I

Chapter 33 – Accessing the Hive MetaStore

As indicated earlier in the book, Hadoop doesn't store its metadata in the Hadoop database itself, but rather in an external database such as MySQL, PostgreSQL, etc. This is in stark contrast (the bleakest of all the contrasts) to other databases, such as Oracle, SQL Server, Teradata, etc. which have metadata readily available in tables such as `ALL_TABLES`, `ALL_TAB_COLUMNS`, `INFORMATION_SCHEMA.TABLES`, `INFORMATION_SCHEMA.COLUMNS`, etc.

As SQL and HPL/SQL programmers, we cannot live all gulag-y like this since the ability to write more generic code depends on knowing whether, say, a specific table exists, or knowing if a column is available within a table, or knowing a column's data type, and so on. Now, there are two ways to mambo here:

1. Metadata via ImpalaSQL – In order to query the metadata in a similar manner used with your legacy database, the metadata needs to be available in one or more tables in the Hadoop database. Think `ALL_TABLES` or `INFORMATION_SCHEMA.TABLES` here. This can be done by creating a Linux script to unload the metadata stored in the external database (e.g., MySQL, Postgres, etc.) and then load that metadata into Hadoop tables for use with ImpalaSQL. At this point, if you're an Oracle programmer, say, you're back to the comfy `ALL_TABLES` and `ALL_TAB_COLUMNS`. A similar comment applies to SQL Server, Teradata, etc. programmers. Naturally, it takes time to perform the unload and load, but the script can be set to run, say, every `10` minutes keeping these two internal metadata tables fresh-*ish*. You may want to play with the refresh rate, but keep in mind that if you have several databases and many tables, the unload and load process may take some time, so set the refresh rate longer than the time it takes for the unload and load process to…uh…process.
2. Metadata via HPL/SQL – To make coding easier, several functions can be created which access the metadata **directly** giving the HPL/SQL programmer immediate and up-to-date metadata information. You can access the external database by using the `hplsql-site.xml` file connection information to the metadata database, cut over to that connection within the HPL/SQL functions, and then switch back to the ImpalaSQL connection at the end of the HPL/SQL functions. This way, you're always accessing the most recent metadata from your HPL/SQL programs.
3. As indicated several times throughout the book, Hive version 3 contains the `sys` database schema accessible via HiveQL, but not ImpalaSQL. Similar to bullet #1 above, the relevant `sys` tables can be copied in a timely manner over to tables where ImpalaSQL can access them.

Note that the external database and its stored information is known generically as the *MetaStore*. *And I never metastore I didn't like! Ba-doom-chee! I'm here all week!* Both Hive and Impala share the same MetaStore which is why issuing an `INVALIDATE METADATA` *`table-name`*`;` in ImpalaSQL allows the tables stored in Hive to be recognized and accessible immediately afterwards from Impala.

Please peruse the relevant response to the Hadoop Administrator E-Mail for connection information to the MetaStore. Note that this chapter is necessarily generic since I don't know which external database your Hadoop Administrator is using. Please work with your biologically-upgraded Hadoop Administrator if/when problems arise. In the examples below, I assume you're using MySQL as the MetaStore.

Familiarizing Yourself with the MetaStore's Metadata Tables

In this section, we'll first look at the tables available in the external MetaStore database. Note that these tables will be very familiar if you've accessed the metadata from your legacy database. Now, you may need to prepend the schema name (say, `hive`) to the metadata tables below (e.g., `hive.DBS`, etc.). I'll leave that out in the discussion below, but you may want to confirm this with your uber-brainy Hadoop Administrator. Also, note that not every column available in the tables shown below is listed, only the ones relevant for this fireside chat.

- `DBS` – This table contains the database names and other information:
  - `DB_ID` (`BIGINT`) – The database identifier
  - `NAME` (`VARCHAR(128)`) – The name of the database
  - `DESC` (`VARCHAR(4000)`) – The description of the database
- `TBLS` – This table contains the table names and other information:
  - `DB_ID` (`BIGINT`) – The database identifier

  - `TB_ID` (`BIGINT`) – The table identifier
  - `SD_ID` (`BIGINT`) – The table-column link identifier
  - `TBL_NAME` (`VARCHAR(128)`) – The name of the table
  - `TBL_TYPE` (`VARCHAR(128)`) – The type of the table
  - `OWNER` (`VARCHAR(767)`) – The name of the owner/creator of the table (e.g., `smithbob`)
- `COLUMNS_V2` – This table contains the column names and other information
  - `CD_ID` (`BIGINT`) – The column identifier
  - `COLUMN_NAME` (`VARCHAR(128)`) – The name of the column
  - `TYPE_NAME` (`VARCHAR(4000)`) – The name of the column's data type
  - `INTEGER_IDX` (`INT`) – The column's location number in the `CREATE TABLE` statement (starts from `0`)
- `SDS` – This table is used to link the tables to their associated columns
  - `SD_ID` (`BIGINT`) – The table-column link identifier (or storage information ID)
  - `CD_ID` (`BIGINT`) – The column identifier
- `PARTITION_KEYS` – Although the table `COLUMNS_V2` contains the column names, data types, and so on, if the table is partitioned, the columns used to partition the table don't appear in `COLUMNS_V2`, but rather they appear in this table.
  - `TBL_ID` (`BIGINT`) – The table identifier
  - `PKEY_NAME` (`VARCHAR(128)`) – The partition key column name (akin to `COLUMNS_V2.COLUMN_NAME`)
  - `PKEY_TYPE` (`VARCHAR(767)`) – The partition key column data type (akin to `COLUMNS_V2.TYPE_NAME`)
  - `INTEGER_IDX` (`INT`) – The column's location number in the `CREATE TABLE` statement (starts from `0`)

For example, to pull a list of the tables in all databases, you can submit a query such as this:

```
SELECT UPPER(D.NAME) AS DATABASE_NAME,
       UPPER(T.TBL_NAME) AS TABLE_NAME,
       T.OWNER AS TABLE_OWNER
 FROM DBS D INNER JOIN TBLS T
 ON D.DB_ID=T.DB_ID
 ORDER BY 1,2;
```

To pull a list of tables as well as column information, you can submit a query such as this (we talk about the homemade column `RESULT_ORDER` further below):

```
SELECT UPPER(D.NAME) AS DATABASE_NAME,
       UPPER(T.TBL_NAME) AS TABLE_NAME,
       C.INTEGER_IDX AS COLUMN_ID,
       UPPER(C.COLUMN_NAME) AS COLUMN_NAME,
       UPPER(T.OWNER) AS TABLE_OWNER,
       UPPER(C.TYPE_NAME) AS DATA_TYPE,
       CAST(1 AS TINYINT) AS RESULT_ORDER
FROM DBS D INNER JOIN TBLS T
 ON D.DB_ID=T.DB_ID
  INNER JOIN SDS S
  ON T.SD_ID=S.SD_ID
   INNER JOIN COLUMNS_V2 C
   ON S.CD_ID=C.CD_ID
 ORDER BY 1,2;
```

With the information above, you may want to have your Hadoop Administrator create one or more database views within the MetaStore database to bring together all of this information. For example, one view could be just table-related (i.e., `ALL_TABLES`, `INFORMATION_SCHEMA.TABLES`) and another view table/column-related (i.e., `ALL_TAB_COLUMNS`, `INFORMATION_SCHEMA.COLUMNS`).

Note that if any of your tables are partitioned – and I'm guessin' that's a big honkin' **YES!** – you'll have to `UNION` the information contained in the table `PARTITION_KEYS` with the information in the table `COLUNNS_V2` so you don't miss out on any of the columns. Also, note that the column `INTEGER_IDX` restarts from `0` in the table `PARTITION_KEYS`, but knowing this factoid means you can build that into a query similar to the SQL fragment shown below:

```
...snip...
SELECT 1 AS RESULT_ORDER,
       INTEGER_IDX,
       COLUMN_NAME,
       TYPE_NAME
 FROM COLUMNS_V2
UNION ALL
SELECT 2 AS RESULT_ORDER,
       INTEGER_IDX,
       PKEY_NAME AS COLUMN_NAME,
       PKEY_TYPE AS TYPE_NAME
 FROM PARTITION_KEYS
ORDER BY 1,2
...snip...
```

With the combination of columns `RESULT_ORDER` and `INTEGER_IDX`, you'll never confuse the order of the columns: just sort by `RESULT_ORDER` and `INTEGER_IDX` and the columns will appear in the same order as defined on the `CREATE TABLE` Statement. Sweet!

Metadata via ImpalaSQL

In this section, we create a Linux script which unloads the table and column metadata from the MetaStore and loads it into the database. First, though, let's create two external tables, one for the tables and one for the tables, columns, etc. Note that you'll have to replace the HDFS directories below with the appropriate directories you've created using `hadoop fs -mkdir`.

```
CREATE EXTERNAL TABLE PROD_SCHEMA.ALL_TABLES(
 DATABASE_NAME STRING,
 TABLE_NAME STRING,
 TABLE_OWNER STRING
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/hdfs-directory/all_tables';

CREATE EXTERNAL TABLE PROD_SCHEMA.ALL_TAB_COLUMNS(
 DATABASE_NAME STRING,
 TABLE_NAME STRING,
 COLUMN_ID SMALLINT,
 COLUMN_NAME STRING,
 DATA_TYPE STRING,
 RESULT_ORDER TINYINT
)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE
LOCATION '/hdfs-directory/all_tab_columns';
```

Note that these two tables can be created from the `impala-shell` command line or from your favorite SQL client up-front since they only have to be created once. It's the underlying data in the `/all_tables` and `/all_tab_columns` HDFS directories that will change.

Next, let's create the Linux script, called `updateMetadata`, which will unload the MetaStore, create two separate tab-delimited files (one for tables and one for tables/columns) and then copy them over to the correct directories. Note that you'll have to replace `mysql` with the appropriate database utility as well as the hostname, username, etc.

```
#!/bin/bash -v

# Bring in the .bash_profile to capture the PATH.
source $HOME/.bash_profile

# Remove the temporary tables.
rm -f $HOME/all_tables.tsv
rm -f $HOME/all_tab_columns.tsv

# Unload the MetaStore for the tables temporarily to $HOME
mysql -hhostname -uusername -ppassword --database=dbname --skip-column-names -e
"SELECT UPPER(D.NAME) AS DATABASE_NAME,UPPER(T.TBL_NAME) AS TABLE_NAME, T.OWNER
AS TABLE_OWNER FROM DBS D INNER JOIN TBLS T ON D.DB_ID=T.DB_ID" >
$HOME/all_tables.tsv

# Unload the MetaStore for the tables/columns temporarily to $HOME
mysql -hhostname -uusername -ppassword --database=dbname --skip-column-names -e
"SELECT UPPER(D.NAME) AS DATABASE_NAME,UPPER(T.TBL_NAME) AS
TABLE_NAME,C.INTEGER_IDX AS COLUMN_ID,UPPER(C.COLUMN_NAME) AS
COLUMN_NAME,UPPER(T.OWNER) AS TABLE_OWNER,UPPER(C.TYPE_NAME) AS
DATA_TYPE,CAST(1 AS TINYINT) AS RESULT_ORDER FROM DBS D INNER JOIN TBLS T ON
D.DB_ID=T.DB_ID INNER JOIN SDS S ON T.SD_ID=S.SD_ID INNER JOIN COLUMNS_V2 C ON
S.CD_ID=C.CD_ID UNION ALL SELECT UPPER(D.NAME) AS
DATABASE_NAME,UPPER(T.TBL_NAME) AS TABLE_NAME,P.INTEGER_IDX AS
COLUMN_ID,UPPER(P.PKEY_NAME) AS COLUMN_NAME,UPPER(T.OWNER) AS
TABLE_OWNER,UPPER(P.PKEY_TYPE) AS DATA_TYPE,CAST(2 AS TINYINT) AS RESULT_ORDER
FROM DBS D INNER JOIN TBLS T ON D.DB_ID=T.DB_ID INNER JOIN SDS S ON
T.SD_ID=S.SD_ID INNER JOIN PARTITION_KEYS P ON T.TBL_ID=P.TBL_ID" >
$HOME/all_tab_columns.tsv

# Copy the local all_tables.tsv over to the HDFS all_tables directory.
# Note: The -f switch forces replacement if the file exists.
hadoop fs -copyFromLocal -f $HOME/all_tables.tsv .../all_tables

# Copy the local all_tab_columns.tsv over to the HDFS all_tab_columns directory
# Note: The -f switch forces replacement if the file exists.
hadoop fs -copyFromLocal -f $HOME/all_tab_columns.tsv .../all_tab_columns

# Now that the files are located in their respective directories, we have
#  to tell Hadoop to recognize that the files have changed.
impala-shell -quiet -i hdpserver -database=prod_schema --query "invalidate
metadata all_tables;invalidate metadata all_tab_columns;refresh
all_tables;refresh all_tab_columns;compute stats all_tables; compute stats
all_tab_columns;"

exit
```

Update the ellipses above to point to the appropriate HDFS directory associated with each table (`all_tables` and `all_tab_columns`).

You can now place `updateMetadata` in the crontab file to run, say, every `10` minutes:

```
*/10 * * * updateMetadata
```

Metadata via HPL/SQL

In this section, we create several HPL/SQL functions which access the MetaStore directly using the connection information in the `hplsql-site.xml` file. First, either you or your star-studded Hadoop Administrator must modify the `hplsql-site.xml` file to add/update the appropriate entry to the MetaStore. For example, below is the `property` branch specific to MySQL with a name of `hplsql.conn.mysqlconn`:

```
<property>
 <name>hplsql.conn.mysqlconn</name>
 <value>
  com.mysql.jdbc.Driver;jdbc:mysql://hostname/dbname;username;password
 </value>
 <description>MySQL connection to the Hive MySQL Metastore</description>
</property>
```

If you're using a different external metadata database, such as PostgreSQL, modify the appropriate section of the `hplsql-site.xml` file to reflect that.

Next, let's create the HPL/SQL function `table_exists` located in the file `table_exists.hplsql` which returns a `1` if the table is found in the database; otherwise, `0` is returned. **Take note that we temporarily connect to the MySQL database, perform a query on the metadata, then connect back to Impala.** Naturally, you can modify this code based on you and/or your Team's requirements.

```
create or replace function table_exists(psTBLNAME in string) return int as

 iCNT int;
 sSQL string;
 sDBNAME string;

 begin

  /* Check if the passed-in table name is null or not */
  if (psTBLNAME is null) then
   return(null);
  end if;

  /* Connect to the MySQL MetaStore */
  set hplsql.conn.default=mysqlconn;

  /* Set the default database: prod_schema */
  sDBNAME := 'prod_schema';

  /* Prepare the SQL code to query the MySQL database */
  sSQL := "select count(*) from dbs d inner join tbls t on d.db_id=t.db_id where
upper(t.tbl_name)='" || upper(psTBLNAME) || "' and upper(d.name)='" ||
upper(sDBNAME) || "'";

  /* Execute the SQL query placing the results of the count into iCNT */
  execute(sSQL) into iCNT;

  /* Reconnect to Impala since we are done with MySQL */
  set hplsql.conn.default=impala;

  /* Return the appropriate return code */
  if iCNT > 0 then
   return(1);
  else
   return(0);
  end if;
```

```
 end;
```

In a similar way, we can create the HPL/SQL function `column_exists` located in the file `column_exists.hplsql` to check that a specific column exists within a table:

```
create or replace function column_exists(psTBLNAME in string,psCOLNAME in string)
                                                            return int as

 iCNT int;
 sSQL string;
 sDBNAME string;

 begin

  /* Check if the passed-in table name is null or not */
  if (psTBLNAME is null) then
   return(null);
  end if;

  /* Check if the passed-in column name is null or not */
  if (psCOLNAME is null) then
   return(null);
  end if;

  /* Connect to the MySQL MetaStore */
  set hplsql.conn.default=mysqlconn;

  /* Set the default database: prod_schema */
  sDBNAME := 'prod_schema';

  /* Prepare the SQL code to query the MySQL database */
  sSQL := "select count(*) from dbs d inner join tbls t on d.db_id=t.db_id inner
join sds s on t.sd_id=s.sd_id inner join columns_v2 c on s.cd_id=c.cd_id where
upper(t.tbl_name)='" || upper(psTBLNAME) || "' and upper(d.name)='" ||
upper(sDBNAME) || "' and upper(c.column_name)='" || upper(psCOLNAME) || "'";

  /* Execute the SQL query placing the results of the count into iCNT */
  execute(sSQL) into iCNT;

  /* Reconnect to Impala since we are done with MySQL */
  set hplsql.conn.default=impala;

  /* Return the appropriate return code */
  if iCNT > 0 then
   return(1);
  else
   return(0);
  end if;

 end;
```

Now, to use these functions in your HPL/SQL code, include the two `.hplsql` files and then call them in the normal manner:

```
include table_exists.hplsql
include column_exists.hplsql

...snip...

iTblExists int;
iColExists int;

...snip...

iTblExists := table_exists(sTableName);
iColExists := column_exists(sTableName,sColumnName);

...snip...
```

Chapter 34 – Working with Impala Request Pools

As you can well imagine, submitting multiple jobs at once could tax your Linux edge node server as well as the Hadoop cluster. Depending on the amount of data you're slogging around as well as how powerful the nodes are, you may see some performance degradation, or even jobs just plain failing. This may be especially apparent if you create a website for your many non-technical users to submit requests at the simple push of a button. One way around this is to work with your Hadoop Administrator to set up *request pools* on the cluster. Request pools are a part of Impala's *Admission Control Feature*.

A request, or resource, pool defines a limitation of services on the cluster such as maximum amount of allowable memory usage, maximum allowable run time, etc. Each request pool is given a friendly name which you can then specify when you submit your job. Note that your user ID will need to have permission to submit jobs to a request pool, something your Hadoop Administrator can easily do.

Your Hadoop Administrator can limit the total number of jobs for each request pool. If a job is submitted via a request pool that's currently at its maximum capacity, that job will wait until a slot opens up, whereupon it begins to execute. Note that your Hadoop Administrator can limit the wait time as well and, if exceeded, your job may be booted from the queue and you'll have to re-submit your job. *Wuh, wuh, wuuuuuuh!*

While you're performing the conversion from your legacy database to the Hadoop database, it's best to avoid dealing with request pools since you already have a lot to worry about as it is. But, once the conversion is complete, and your team members are submitting jobs, you may want to revisit this topic and have a heart-to-heart with your magnificent Hadoop Administrator about it.

Recommended Request Pools

In this section, we recommend four request pools. Unfortunately, we cannot recommend the settings (i.e., max memory, max runtime, etc.) for each pool since that depends on your cluster and job execution profiles. Please work with your Hadoop Administrator to determine the most appropriate settings for each request pool.

- ☐ `hdpserver_small_pool` – This request pool is used for smaller jobs with minimal memory and runtime limits. For example, a maximum of, say, 10 jobs will run in parallel with this request pool.
- ☐ `hdpserver_medium_pool` – This request pool is used for medium-sized jobs and allows for more memory and longer runtime limits. For example, a maximum of, say, 5 jobs will run in parallel with this request pool.
- ☐ `hdpserver_large_pool` – This request pool is used for large jobs and allows for much more memory and much longer runtime limits. For example, a maximum of, say, 2 jobs will run in parallel with this request pool.
- ☐ `hdpserver_maximum_pool` – This request pool is used for very large jobs and has the largest memory and runtime limits. This request pool is, say, for production jobs running at night when everyone is in bed playing Wordle.

Selecting a Request Pool

There are several ways to select a request pool.

If you're using the Linux command line utility `impala-shell`, you can easily select a request pool by entering the following while at the `impala-shell` command line:

```
set request_pool=hdpserver_small_pool;
```

If you're submitting SQL code with the `-f` switch, ensure the line above appears near the top of your code. On the other hand, if you're submitting SQL code with the `-q` switch, ensure the line above appears in the quoted string first followed by your SQL query.

From an HPL/SQL program, you can use the `EXECUTE()` function to set the request pool, most likely near the top of the procedure:

```
execute("set request_pool=hdpserver_small_pool;");
```

If you're using Toad Data Point or other SQL client, you can specify the request pool within the connection string itself:

```
Driver=Cloudera ODBC Driver for Impala;...;ssp_request_pool=hdpserver_small_pool;
```

Please check the documentation for your driver to determine the correct option name to set.

Chapter 35 – Making a Backup Copy of a Linux Directory

Occasionally, and out of a complete and utter sense of paranoia, it may be a good idea to back up your entire Linux account (for example, Bob's `/home/smithbob`) as well as other important directories. One way to do this is to use FTP software, such as FileZilla, to copy files from the Linux edge node server to your laptop and then, maybe, to a backup drive or corporate cloud storage location. Another way is to use `tar` (tape archive) to backup an entire folder and all of its directories and contents and then transfer that single large file over to your laptop, backup drive, cloud storage, etc. In this chapter, we'll talk about how to use `tar` to copy an entire directory.

Please have a conversation with your Linux Administator about any backup procedures performed on the Linux edge node server. If your server is not being backed up on a timely basis, you may want to press the Linux Administrator for it. And, don't forget to send your Linux Administrator a lovely gift basket.

Backing Up a Directory Using `tar`

Using PuTTY, log into the Linux edge node server and then change directory to the `/tmp` folder:

```
cd /tmp
```

This folder tends to have a large amount of temporary storage available to use. Next, run the following command changing where appropriate:

```
tar --warning=no-file-changed -zvcf lnxserver_smithbob.tgz /home/smithbob
```

This command will create a gzip'd tar file named `lnxserver_smithbob.tgz` which will contain the entire contents of the `/home/smithbob` directory as well as all subdirectories and contents. Note that the option `--warning=no-file-changed` will prevent you from receiving a warning if one of the files changed during the process of the `tar` operation.

Once `tar` has completed, you can move the file `lnxserver_smithbob.tgz` back to your home folder:

```
mv lnxserver_smithbob.tgz $HOME
```

At this point, you can FTP the file `lnxserver_smithbob.tgz` to your laptop or other storage medium for safe-keeping. Phew!!

Chapter 36 – Using `ssh` and `scp` from Linux and Windows

Throughout the book, our entire world has been the Linux edge node server as well as your laptop. But, your network topology is probably much more complex than that and probably includes multiple Linux servers, database servers (Hadoop or otherwise), cloud computing servers, Nespresso machines, Farming Simulator game servers, and so on.

Up to now, we've used PuTTY to log into the Linux edge node server to interact with the operating system to, say, submit SQL queries using `impala-shell`, run HPL/SQL programs, and so on. We've also used FileZilla to FTP files between the Linux edge node server and our laptop. With a complex network topology, PuTTY and FileZilla aren't the only way to go, Joe!

In this chapter, we'll look at the *secure shell* command (`ssh`), which allows you to log into the Linux edge node server from either a Linux or Windows command line. We'll also describe how to work with the *secure copy* (`scp`) command, which allows you to transfer files between servers. Both are run from the command line only which means no GUI interface, but they lend themselves to scripting. I'll take that over a GUI any day!

Secure Shell (`ssh`)

Secure shell (`ssh`) is similar to PuTTY except there's no GUI interface. You can use `ssh` from Windows as well as Linux. From Windows, you can log into the Linux edge node server as an alternative to PuTTY. While logged into the Linux edge node server, you can further use `ssh` to log into the same or another Linux server. For example, while you're logged into your home account (`/home/smithbob`, say) on the Linux edge node server, you can use `ssh` to log into the production account to execute some production code from the command line there. Once you log out, you're back in your home account again. But, even better, you can use `ssh` to log into a remote server to perform some tasks. Even betterer, you can tell `ssh` to run commands on the remote server from the `ssh` command line itself without actually having to log into the remote server's command line. Diabolical!!

The Windows operating system should have a version of secure shell, called OpenSSH, installed by default. If not, you can install Cygwin, outlined in *Appendage #2 – Linux on Windows*, along with OpenSSH to gain the same functionality. Regardless of the why's and wherefore's, the command is named `ssh`.

At its very simplest, the syntax for `ssh` allows a user named *`username`* to log into a remote server named *`remote_machine`*:

```
ssh username@remote_machine
```

For example, from the Windows Command Prompt, let's have Bob (`smithbob`) log into the Linux edge node server (`lnxserver`):

```
ssh smithbob@lnxserver
```

Similar to PuTTY, `ssh` may ask you if you want to continue to connect to the remote server:

```
The authenticity of host 'lnxserver (10.20.30.40)' can't be established.
ECDSA key fingerprint is SHA256:a7VrQZuy0jjufLNfpYtwixbarWiYBous36ARKOMtBsF.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

At the prompt, type `yes` and hit the Enter key. You'll only be asked this once for each server you log into with `ssh`. You'll see something like the following and you'll then be asked to enter your password.

```
Warning: Permanently added 'lnxserver,10.20.30.40' (ECDSA) to the list of known
hosts.
smithbob@lnxserver's password:
```

At this point, you're logged in to the remote server ready to *command up a storm*!

The general syntax for `ssh` is a blinding stream of options:

```
usage: ssh [-46AaCfGgKkMNnqsTtVvXxYy] [-B bind_interface]
           [-b bind_address] [-c cipher_spec] [-D [bind_address:]port]
           [-E log_file] [-e escape_char] [-F configfile] [-I pkcs11]
           [-i identity_file] [-J [user@]host[:port]] [-L address]
           [-l login_name] [-m mac_spec] [-O ctl_cmd] [-o option] [-p port]
           [-Q query_option] [-R address] [-S ctl_path] [-W host:port]
           [-w local_tun[:remote_tun]] username@remote_server [command]
```

If you'd like to run a command directly on the remote server, place the command after *`username@remote_server`*. For example, from the Windows Command Prompt, let's get a directory listing of `/home/smithbob` on the remote server `lnxserver` using the Linux command `ls -alF`:

```
ssh smithbob@lnxserver ls -alF

total 25
drwxr-xr-x+ 1 smithbob None    0 Feb 28 13:02 ./
drwxrwxrwt+ 1 smithbob None    0 Feb 27 17:12 ../
-rw-------  1 smithbob None  218 Feb 28 14:06 .bash_history
-rwxr-xr-x  1 smithbob None 1494 Feb 27 17:09 .bash_profile*
-rwxr-xr-x  1 smithbob None 5645 Feb 27 17:09 .bashrc*
-rwxr-xr-x  1 smithbob None 1919 Feb 27 17:09 .inputrc*
-rwxr-xr-x  1 smithbob None 1236 Feb 27 17:09 .profile*
drwx------+ 1 smithbob None    0 Feb 28 13:37 .ssh/
```

Now, issuing more complicated commands can be…uh…complicated, so the best thing is to create a script on the remote server and then run that script using the ***`command`*** via `ssh`. Unthinkable!!

Note that each time you use `ssh`, you'll have to enter in your password at some point. But, see the section below for instructions on how to workaround this nightmarishly outlandishly vicious issue.

Secure Copy (`scp`)

Similar to FileZilla, you can use `scp` to copy files from one machine to another. At its simplest, to pull a file named `pull_this_file.txt` (located in Bob's home directory `/home/smithbob`, say) from the remote server (`lnxserver`, say) to our Windows laptop, issue the following command from the Windows Command Prompt (or Cygwin Terminal):

```
scp smithbob@lnxserver:/home/smithbob/pull_this_file.txt .
```

The single dot at the end of the command indicates, as you already know, the current working directory; thus, the file `pull_this_file.txt` is copied over to whatever directory on Windows you happened to have landed in at the time.

The general syntax for `scp` is as follows:

```
usage: scp [-346BCpqrTv] [-c cipher] [-F ssh_config] [-i identity_file]
            [-J destination] [-l limit] [-o ssh_option] [-P port]
            [-S program] source target
```

Note that either *`source`* or *`target`* can take the form *`username@remote_server`* depending on where you're doing the pushing and pulling. Take note that you indicate a file location and name after a colon, as shown in the example above. For example, to push a file named `borscht_recipe.txt` from your Windows laptop to the remote server, you can do something like this at the Windows Command Prompt:

```
scp C:\borscht_recipe.txt smithbob@lnxserver:/home/smithbob/borscht_recipe.txt
```

Similar to `ssh`, `scp` requires that you be physically available and somewhat *compos mentis* to enter in your password. But, see the next section.

Note that `scp` display information while your file is being transferred. For example,

```
borscht_recipe.txt                                    92%   45KB   2.9MB/s   00:05
```

In this example, the columns to the right indicate the following:

- `92%` – indicates the percentage of the file's body parts that have been transferred over
- `45KB` – indicates the associated kilobytes transferred over
- `2.9MB/s` – indicates the speed of the transfer
- `00:05` – indicates the elapsed time (you may also see the acronym ETA while the transfer is occurring)

Using ssh and scp without a Password

If you want to use `ssh` and `scp` without having to enter in a password at the terminal, follow the instructions below. There must be two consenting adult servers involved: one that's doing the logging in (which I'll call the *local machine*) and the other that accepts the login (which I'll call the *remote machine*); that is, one computer issues the `ssh`/`scp` command and the other is being logged into.

Step #1: Generate a Public and Private Key Pair

At the *local machine*'s command prompt, you'll have to generate a public and private key pair. Issue the following from the command prompt:

```
ssh-keygen -t rsa
```

The output from this command will be similar to the following (some output silliness removed):

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/smithbob/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/smithbob/.ssh/id_rsa.
Your public key has been saved in /home/smithbob/.ssh/id_rsa.pub.
The key fingerprint is:
A9:54:f1:21:fa:98:41:da:ba:05:8d:51:2d:10:e5:8f
smithbob@lnxserver
```

```
* Your private key is in a file named ~/.ssh/id_rsa
* Your public key is in a file named ~/.ssh/id_rsa.pub
```

Note that you're requested to create a passphrase. Avoid using your account's password, but instead provide some text that only you'll remember and no one else knows (except, maybe, for your significant other, your relatives, their friends, your barber, the building security guard, your car mechanic, and so on).

Take note of the two **emboldened** lines above. Your *private key* is located in the file `id_rsa` and your *public key* is located in the file `id_rsa.pub`. Note that the tilde (~) just indicates the `$HOME` directory, which is `/home/smithbob`, in our example.

NEVER GIVE YOUR PRIVATE KEY TO ANYONE! THE `id_rsa` FILE WILL STAY RIGHT WHERE IT IS, THANK YOU VERY MUCH. AS FOR YOUR PUBLIC KEY (`id_rsa.pub`), YOU'RE GOING TO COPY THAT BAD BOY

TO YOUR *REMOTE MACHINE* IN THE FOLLOWING STEP. APOLOGIES FOR TYPOGRAPHICALLY SCREAMING AT YOU, POPPETS!

Step #2: Copy your Public Key to your Remote Server

The next step is to copy the **`id_rsa.pub`** file to the *remote machine* you want to be able to access with `ssh`/`scp` without using a password.

Next, log into your *remote machine*, which could be `lnxserver` or another server. If the `.ssh` directory does not already exist in you home directory, create it in the usual manner:

```
mkdir .ssh
```

Don't forget that files beginning with a period are normally hidden when using the Linux command `ls`. But, if you've created the alias `lsf` for `ls -alF`, then you'll see the hidden files (due to the `-a` switch).

Now, on the local machine, copy the `id_rsa.pub` file to a new file named `authorized_keys` in the `.ssh` directory. Simply transfer the file `id_rsa.pub` from the *local machine* over to the *remote machine*.

Log out of the *remote machine*.

Assuming all of these steps worked without a hitch, you're now ready to test your *remote machine* login without using a password from the *local machine*.

Step 3: Using `ssh`/`scp` without a Password

You can now use both secure shell (`ssh`) and secure copy (`scp`) from the *local machine* without using a password to connect to the *remote machine*. When you issue either command, you shouldn't be asked for a password.

Note that if your password changes due to some silly corporate policy or other inane reason, you don't have to update the public key file since these keys are not linked to your password (which, as we all know, is the name of your first born child followed by an exclamation point).

Chapter 37 – The Linux `/etc/skel` Directory

As we discussed in *Chapter 25 – Introduction to HPL/SQL*, you can ask your Linux Administrator to locate both the `hplsql` utility and the `hplsql-site.xml` driver file in one location and update both, if necessary, there.

Alternatively, you can locate both files in a directory under each user's account, such as `/home/smithbob/hplsql` (where `hplsql` here is the name of the directory and not the utility). This allows each user to update their own copy of both files, if necessary. Unfortunately, with this second option, you're going to have to add these files **by hand for each of the potentially thousands of users on your Team!** Nah…just kidding…

When your Linux Administrator adds a new user, each new account is created with a basic set of files such as the all-important `.bash_profile` file, the somewhat important `.bashrc` file, the unimportant `this_is_an_unimportant_file` file, etc. as well as a few more files. But, be aware that your Linux Administrator can include additional files or folders when creating a new user account by adding them to the Linux `/etc/skel` directory. This directory can include other directories, such as the `hplsql` directory containing both the `hplsql` utility and `hplsql-site.xml` driver file.

For example, the default `/etc/skel` directory for my CentOS 8 machine looks like this:

```
[smithbob@lnxserver ~]$ lsf /etc/skel
total 28
drwxr-xr-x.   3 root root    78 Dec 31 15:10 ./
drwxr-xr-x. 175 root root 12288 Feb 28 08:53 ../
-rw-r--r--.   1 root root    18 Jul 27  2021 .bash_logout
-rw-r--r--.   1 root root   141 Jul 27  2021 .bash_profile
-rw-r--r--.   1 root root   376 Jul 27  2021 .bashrc
drwxr-xr-x.   4 root root    39 Dec 31 15:10 .mozilla/
[smithbob@lnxserver ~]$
```

Please have a conversation with your Linux Administrator about including additional content to `/etc/skel` if you decide to go down this route (or for any other reason). Just ensure that any files that are meant to be executed, such as `hplsql`, are set as executable in the `/etc/skel` directory.

Now, when a new user is added by your Linux Administrator, the entire contents of the `/etc/skel` directory is used to create the user's `/home` directory, `hplsql` folder and all. Naturally, you can ask your Linux Administrator to add more directories, scripts, utilities, etc. so that any new user is immediately able to *hit the ground running*. Even that unpaid summer intern! Amazing!!

Chapter 38 – The `parquet-tools` and `parquet-cli` Utilities

In the section labeled *Parquet Viewer* in *Chapter 3 – Recommended Windows Client Software*, we showed you how to install a Windows application which allows you to view data in a file stored in Parquet format. Another way to do a similar thing is to use the Linux command line utilities `parquet-tools` and `parq`. Note that one or both of these tools may not be installed by default. After reading this chapter, ask your top-notch Hadoop Administrator to install one or both of these tools for you on the Linux edge node server.

The command line utility `parquet-tools` allows you to view the data as well as the schema of a Parquet file. There are two ways to use `parquet-tools`:

- ☐ Linux File System – If the Parquet file is located on the Linux file system, say the directory `/home/smithbob`, you can use `parquet-tools` directly on the file:

```
[smithbob@lnxserver ~]$ parquet-tools cat dim_calendar.parq
```

- ☐ HDFS – If the Parquet file is located in HDFS, you'll have to use the command `hadoop jar` along with the appropriate Java `.jar` file in order to access the file in HDFS. On my system, the available Java `.jar` file is `/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/jars/parquet-tools-1.10.99.7.1.7.0-551.jar`, but your file may be named something even more wild. Ask your lively Hadoop Administrator about it! For example, to display the Parquet file associated with the table `dim_calendar` using `parquet-tools`, you can execute something like the following at the Linux command prompt (on one line, please!):

```
[smithbob@lnxserver ~]$ hadoop jar
          /opt/cloudera/.../parquet-tools-1.10.99.7.1.7.0-551.jar
          cat
          hdfs://lnxserver.com:8020/.../7d4ef2af478388ef-5c3a50e700000000_2110153450_data.0.parq
```

Recall that you can obtain the `Location` of the underlying Parquet file(s) associated with the table `dim_calendar` by issuing `desc formatted dim_calendar;` from Impala.

Unfortunately, when using `parquet-tools`, text strings are displayed in their mysterious internal format:

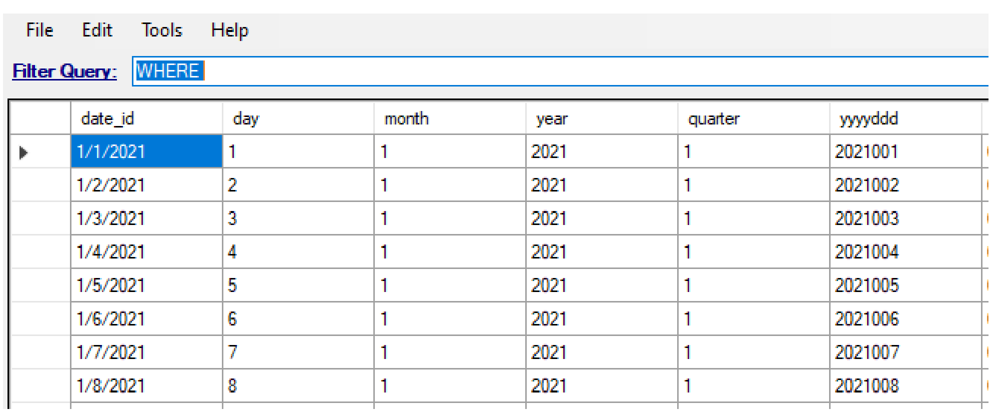
```
date_id = 18628
day = 1
month = 1
year = 2021
quarter = 1
yyyyddd = MjAyMTAwMQ==
ddd = MDAx
first_day_of_month = 18628
first_day_of_quarter = 18628
first_day_of_year = 18628
month_name = SmFudWFyeQ==
weekday_name = RnJpZGF5
yyyyqq = MjAyMTAx
yyyymm = MjAyMTAx
yyyymmdd = MjAyMTAxMDE=
date_long = SmFudWFyeSAwMSwgMjAyMQ==
date_short = MDFKQU4yMDIx
```

- ☐ `parquet-cli` – One way around the garbled text is to use the Python program `parquet-cli` which can be run using the Linux command `parq`. Note that `parq` doesn't access HDFS, unlike `parquet-tools`, so you'll have to pull the file from the HDFS world to the Linux world. Assuming you copied `dim_calendar` to your account, you can run `parq` on it:

```
[smithbob@lnxserver ~]$ parq dim_calendar.parq --head
      date_id  day  month  year  quarter  yyyyddd  ddd first_day_of_month  \
0  2021-01-01    1      1  2021        1  2021001  001         2021-01-01
1  2021-01-02    2      1  2021        1  2021002  002         2021-01-01
2  2021-01-03    3      1  2021        1  2021003  003         2021-01-01
3  2021-01-04    4      1  2021        1  2021004  004         2021-01-01
4  2021-01-05    5      1  2021        1  2021005  005         2021-01-01
5  2021-01-06    6      1  2021        1  2021006  006         2021-01-01
6  2021-01-07    7      1  2021        1  2021007  007         2021-01-01
7  2021-01-08    8      1  2021        1  2021008  008         2021-01-01
8  2021-01-09    9      1  2021        1  2021009  009         2021-01-01
9  2021-01-10   10      1  2021        1  2021010  010         2021-01-01

  first_day_of_quarter first_day_of_year month_name weekday_name  yyyyqq  \
0           2021-01-01        2021-01-01    January       Friday  202101
1           2021-01-01        2021-01-01    January     Saturday  202101
2           2021-01-01        2021-01-01    January       Sunday  202101
3           2021-01-01        2021-01-01    January       Monday  202101
4           2021-01-01        2021-01-01    January      Tuesday  202101
5           2021-01-01        2021-01-01    January    Wednesday  202101
6           2021-01-01        2021-01-01    January     Thursday  202101
7           2021-01-01        2021-01-01    January       Friday  202101
8           2021-01-01        2021-01-01    January     Saturday  202101
9           2021-01-01        2021-01-01    January       Sunday  202101

   yyyymm  yyyymmdd          date_long date_short
0  202101  20210101  January 01, 2021  01JAN2021
1  202101  20210102  January 02, 2021  02JAN2021
2  202101  20210103  January 03, 2021  03JAN2021
3  202101  20210104  January 04, 2021  04JAN2021
4  202101  20210105  January 05, 2021  05JAN2021
5  202101  20210106  January 06, 2021  06JAN2021
6  202101  20210107  January 07, 2021  07JAN2021
7  202101  20210108  January 08, 2021  08JAN2021
8  202101  20210109  January 09, 2021  09JAN2021
9  202101  20210110  January 10, 2021  10JAN2021
```

Although not the most gorgeous output to look at, at least the text isn't in Martian!

- Your Windows Laptop – Of course, you can view the file on your laptop using the Windows application Parquet Viewer, as discussed earlier:



`parquet-tools cat` *`input`*	Displays all of the rows in the *`input`* Parquet file. If the Parquet file is in HDFS and not on the Linux File System, use the `hadoop jar` command shown above with the appropriate Java `.jar` file.
`parquet-tools head` *`input`*	Displays the first `5` records in the *`input`* Parquet file.
`parquet-tools schema` *`input`*	Displays the schema of the *`input`* Parquet file. This is similar to describing a table in SQL.
`parq` *`input`* `--head`	Displays the first `5` records in the *`input`* Parquet file.
`parq` *`input`* `--tail`	Displays the last `5` records in the *`input`* Parquet file.
`parq` *`input`* `--schema`	Displays the schema of the *`input`* Parquet file.

Although not discussed above, `parquet-tools` does accept `merge` as a command which allows you to append multiple files in Parquet format into one large Parquet file. You may want to avoid this for use with Hadoop. The `merge` command simply appends the Parquet files together **without** attempting to reorganize the data into a more efficient layout resulting in less than optimal query performance.

PART VIII - Advanced Topics II

Chapter 39 – Quick Start Guide to Java Programming

In this admittedly very long chapter, we start off with a quick start guide to Java programming and then move on to user-defined functions (UDFs) programming for ImpalaSQL in *Chapter 40 – Creating User-Defined Functions (UDFs) for ImpalaSQL*.

[Note #1: The author isn't a professional Java programmer, but can hold his own in a pub fight.]
[Note #2: The word "Quick" in the title of this chapter is debateable.]

Java Type System

Each variable declared in Java must be associated with a particular data type. Java breaks up its data types into two distinct groups: *primitives* and *objects*. The primitive data types, such as integers and floating-point numbers, are probably familiar to you if you've been programming for any length of time. A list of primitive data types is shown in the table below:

DATA TYPE KEYWORD	DESCRIPTION	SIZE (in bits)	RANGE
`boolean`	The values `true` or `false`	N/A	`true/false`
`byte`	A 2's-complement integer	8 bits	`-128 to +127`
`short`	A 2's-complement integer	16 bits	`-32768 to +32767`
`int`	A 2's-complement integer	32 bits	`-2,147,483,648 to +2,147,483,647`
`long`	A 2's-complement integer	64 bits	`-9223372036854775808 to +9223372036854775807`
`char`	An unsigned integer representing a UTF-16 code unit	16 bits	N/A
`float`	An IEEE-754 floating-point number	32 bits	7 significant digits
`double`	An IEEE-754 floating-point number	64 bits	15 significant digits

Object data types are discussed later in this chapter.

You declare a variable by entering the data type followed by the name of the variable, such as:

```
int iCounter;
iCounter=1;
```

...or equivalently...

```
int iCounter = 1;
```

Java variable names can start with a letter, underscore (_) or dollar sign (`$`) followed by letters or numbers. Special symbols such as the `@`-sign are usually reserved for Java and should probably be avoided in your own variable names.

Making Comments…Just Sayin'!

You should make comments in your code to remind you what you did or to aid the programmer who takes over your program. There are three types of comments in Java: single-line, multi-line and Javadoc comments.

A single line comment starts with a double-slash (`//`) followed by your comment. Java ignores everything else on the line following the double-slash:

```
int iCounter = 1; //My counter variable set to 1.
```

A multiline comment starts with `/*` and ends with `*/`. Everything in between is ignored even if the comment spans multiple lines:

```
/* This program is used to do something
wonderful and people will be amazed at
how great it is! */ int iCounter=1;
```

The final type of comment is the Javadoc comment and is used to produce API documentation for each variable, method, class, etc. you define in your program. For example,

```
/**
 * iCounter keeps track of counting things.
 *
 */
int iCounter=1;
```

When the documentation is produced, this comment will be associated with the variable `iCounter`.

Conditional Execution

You can use the `if-then-else` or `switch` statements as well as the ternary operator to conditionally execute code based on a condition:

The syntax for a variety of `if-then-else` is as follows:

```
if (condition)
 statement;
```

...or...

```
if (condition)
 statement-1;
else
 statement-2;
```

...or...

```
if (condition) {
 statement-1;
 statement-2;
 ...
}
```

...or...

```
if (condition) {
 statement-1;
 statement-2;
 ...
}
else {
 statement-3;
 statement-4;
 ...
}
```

...or...

```
if (condition-1) {
 statement-1;
 statement-2;
```

```
    ...
  }
  else if (condition-2) {
    statement-3;
    statement-4;
    ...
  }
  else if (condition-3) {
    statement-3;
    statement-4;
    ...
  }
```

...or...

```
  if (condition-1) {
    statement-1;
    statement-2;
    ...
  }
  else if (condition-2) {
    statement-3;
    statement-4;
    ...
  }
  else if (condition-3) {
    statement-3;
    statement-4;
    ...
  }
  else {
    statement-5;
    statement-6;
    ...
  }
```

The syntax for `switch` is as follows:

```
  switch(expression) {
    case constant-1:
      statement-1;
      statement-2;
      ...
      break;
    case constant-2:
      statement-3;
      statement-4;
      ...
      break;
    ...
    default:
      statement-1;
      statement-2;
      ...
      break;
  }
```

The ternary operator is just a short-and-sweet `if-then-else` statement:

```
  conditional-true-false-test ? condition-is-true : condition-is-false;
```

For example,

```
int iMinLenWid = iLength < iWidth ? iLength : iWidth;
```

Looping Constructs

Java has the traditional looping constructs such as the `for`, `while`, and `do-while` loops.

The syntax for the `for` loop is:

```
for (initialize; stopping-condition; increment) {
 statements;
}
```

For example,

```
for (int i=0; i<10; i++) {
 iTotal += i;
}
```

The syntax for the `while` loop is:

```
while (condition) {
 statements;
}
```

For example,

```
while (i<10) {
 iTotal += i;
 i++;
}
```

The `do-while` loop is similar to the `while` loop except that it will execute at least once. The `while` loop, depending on its associated condition, may not execute at all. The meanie!! For syntax for the `do-while` loop is:

```
do {
 statements;
} while (condition);
```

For example,

```
do {
 iTotal += i;
 i++;
} while (i<10);
```

Note that all three of these constructs can make use of the `break` or `continue` statements. The `break` statement will stop a loop from executing immediately upon being reached. The `continue` statement will force the code to jump to the top of the loop skipping all of the code below it.

```
while (i<10) {
 iTotal += i;
    i++;
    if (i==5) {
      break;
    }
}
```

Arithmetic Operators

Java has the traditional arithmetic operators such as + (addition), − (subtraction), * (multiplication), / (division) and % (modulus). Be careful when performing division! If the two operators are integers, then your result will be an integer with any fractional part discarded.

Assignment Operators

Java has the traditional assignment operators such as +=, -=, *= and /=. You can, of course, just use a single equal sign (=) to mean assignment. For example,

```
iCounter += x;
```

is equivalent to

```
iCounter = iCounter + x;
```

Note that you can also string together assignments, such as

```
iCounter = iCounter2 = iCounter3 = 0;
```

Java also has the traditional increment and decrement operators such as x++, ++x, x-- and --x.

Comparison Operators

Java has the traditional comparison operators such as == (is equal to), != (is not equal to), < (is less than), > (is greater than), <= (is less than or equal to) and >= (is greater than or equal to).

Logical Operators

Java has the traditional logical operators such as && (Logical AND), || (Logical OR) and ! (negation). Note that these are logical operators and **not** bitwise operators (we discuss those next).

Bitwise Operators

Java has the traditional bitwise operators such as << (left shift), >> (right shift), >>> (right shift, zero fill), ~ (complement), & (bitwise AND), | (bitwise OR) and ^ (bitwise XOR).

Working with Strings

Besides the char data type, you can use the String data type (which is not primitive, but an object data type) to allow you to work with large strings of text. For example,

```
String sGreeting = "Bonjour";
String sTitle = "Monsieur";
```

You can concatenate two strings together using the + operator:

```
String sFullGreeting = sGreeting + " ," + sTitle;
```

Working with Arrays

Whereas a variable stores one piece of information of a specific data type, an array stores several pieces of information all with a specific data type. For example,

```
int[] aNums = new int[10];
```

The code above creates an array that will hold `10` integers. The empty brackets to the left indicate that `aNums` is an array of integers whereas the `10` on the right indicates how many items the array can hold in total.

You can access a specific array element's value by referring to the array name followed by the element's index number in brackets:

```
int X = aNums[5];
```

Arrays work super-yummy within `for`-loops:

```
for(i=0; i<10; i++) {
 iTotal += aNums[i];
}
```

Exceptions

Rather than using an `if-then-else` construct to catch an exception, you can use the more modern – and dare I say exciting! – `try-catch-finally` block to handle exceptions. Here's what this looks like:

```
class jpgm6 {

 public static void main(String args[]) {

  try {

   //Divide by zero
   int iNum = 5/0;

  }
  catch(ArithmeticException e) {
   System.out.println("Arithmetic Exception Detected: " + e);
  }
  catch(Exception e) {
   System.out.println("Generic Exception Detected: " + e);
  }
  finally {
   System.out.println("Finally!");
  }

 }

}
```

As you see, you start off with the `try` block which is responsible for processing your desired code such as connecting to a database, downloading an HTML web page, computing a value, etc. Next, you provide one or more `catch` blocks whose argument is either the name of a specific exception (such as `ArithmeticException`) or a generic exception (such as `Exception`), and whose body either attempts a retry or notify the user of the exception. Note that `Exception` should appear last in the list of `catch` blocks! The `finally` block will always be executed regardless. Below is a list of some of Java's runtime exceptions you can catch:

```
AnnotationTypeMismatchException, ArithmeticException, ArrayStoreException,
BufferOverflowException, BufferUnderflowException, CannotRedoException,
CannotUndoException, ClassCastException, CMMException,
ConcurrentModificationException, DataBindingException, DOMException,
EmptyStackException, EnumConstantNotPresentException, EventException,
IllegalArgumentException, IllegalMonitorStateException,
IllegalPathStateException, IllegalStateException, ImagingOpException,
IncompleteAnnotationException, IndexOutOfBoundsException, JMRuntimeException,
LSException, MalformedParameterizedTypeException, MirroredTypeException,
MirroredTypesException, MissingResourceException, NegativeArraySizeException,
NoSuchElementException, NoSuchMechanismException, NullPointerException,
ProfileDataException, ProviderException, RasterFormatException,
RejectedExecutionException, SecurityException, SystemException,
TypeConstraintException, TypeNotPresentException, UndeclaredThrowableException,
UnknownAnnotationValueException, UnknownElementException, UnknownTypeException,
UnmodifiableSetException, UnsupportedOperationException, WebServiceException
```

Object-Oriented Programming Concepts

Primitive data types are nice, but they only get you so far. Java is an object-oriented language like C++ and C# and allows you to step up your programming to the next level.

When you program using structured programming, you create a series of functions, subroutines, global variables, local variables, etc. that will help you achieve your desired programming results.

With object-oriented programming (OOP), you create one or more classes (indicated by the `class` keyword) representing objects. Within each class, you have your functions/subroutines (called *methods* in OOP terminology) and variables (called *attributes* in OOP terminology).

For example, you can think of a car as an object. A car has attributes such as exterior color name, number of cylinders, and so on. A car has methods such as the `start_engine` method and the `turn_on_radio` method. Here's an example class for our car:

```
class Car {
 String exteriorColor;
 int numberOfCylinders;
 boolean start_engine() { ...code to start the engine... }
 boolean turn_on_radio() { ...code to turn on the radio... }
}
```

Now, the `Car` class is just a definition. Just like using primitive data types, you have to create a variable that uses the class. For example, to create a usable `Car`, we *instantiate* it using the `new` keyword:

```
Car MyCar = new Car();
```

The code above creates a usable `Car` object called `MyCar` based on the `Car` class.

Now, our class, as defined, doesn't do much of anything. For example, we have no way to set the `exteriorColor` or `numberOfCylinders` for our car. We can change that by adding a *constructor*; that is, a method called whenever the `new` keyword is used to create an object. This constructor allows us to initialize our attributes at instantiation time. For example,

```
class Car {

 String exteriorColor;
 int numberOfCylinders;

 //Our constructor is below
 Car() {
  exteriorColor = "LimeGreen";
  numberOfCylinders = 5;
 }

 boolean start_engine() { ...code to start the engine... }
 boolean turn_on_radio() { ...code to turn on the radio... }
}
```

Now, a constructor MUST have the same name as the class. Note that the constructor above takes no parameters and we force our `exteriorColor` to `LimeGreen` and the `numberOfCylinders` to `5`. However, if you want to pass in either a different exterior color or number of cylinders into the constructor during object instantiation, you'll have to add parameters to your constructor:

```
class Car {

 String exteriorColor;
 int numberOfCylinders;

 //Our constructor is below
 Car(String pExtClr,int pNumCyl) {
  exteriorColor = pExtClr;
  numberOfCylinders = pNumCyl;
 }

 boolean start_engine() { ...code to start the engine... }
 boolean turn_on_radio() { ...code to turn on the radio... }
}
```

Here's how we would instantiate our new car object now:

```
Car MyCar = new Car("LimeGreen",5);
```

Now, `MyCar` is an instantiated object of class `Car` that is `LimeGreen` and with a dubious number of cylinders, `5`. One small problem: we have no way of returning the exterior color or number of cylinders if you need to know them later on in the program. So, let's add two additional methods that will return these two attributes:

```
class Car {

 String exteriorColor;
 int numberOfCylinders;

 //Our constructor is below
 Car(String pExtClr,int pNumCyl) {
  exteriorColor = pExtClr;
  numberOfCylinders = pNumCyl;
 }

 public String getExteriorColor() {
  return(exteriorColor);
 }

 public int getNumberOfCylinders() {
  return(numberOfCylinders);
```

```
 }

 boolean start_engine() { ...code to start the engine... }
 boolean turn_on_radio() { ...code to turn on the radio... }
}
```

These two methods are known as *getters* because they return (or *get*) some piece of information held in the instantiated object. In our case, we're returning the name of the car's exterior color as well as the engine's number of cylinders.

Now, we can use `MyCar` to retrieve both the exterior color and number of cylinders by following the object name with a period and the name of the method:

```
Car MyCar = new Car("PurplePassion",4);
System.out.println(MyCar.getExteriorColor());
System.out.println(MyCar.getNumberOfCylinders());
```

Note that the function `System.out.println()` prints text to the console. We'll look into this more later on.

We can also create two methods that will allow us to update (or *set*) these two attributes. These methods are called *setters*:

```
public void setExteriorColor(String pExtClr) {
 exteriorColor = pExtClr;
}

public void setNumberOfCylinders(int pNumCyl) {
 numberOfCylinders = pNumCyl;
}
```

We can now set the color, say, of our car like this:

```
Car MyCar = new Car("PurplePassion",4);
MyCar.setExteriorColor("DubiousFuchia");
```

Notice that we made use of the keyword `public` for both the getters and the setters. This keyword indicates that, if you have access to the instantiated variable `MyCar`, the outside world can run these four methods. If these four methods used the keyword `private` instead of `public`, then the outside world wouldn't be able to run these methods. For getters and setters, you most likely want the outside world to access them. But, some *methods* may be for **internal use** to the class only and should be set to `private`. Also, some *attributes* may be for internal use to the class only and should be set to `private` as well. For example, if your class makes use of the American Social Security Number (SSN), then you probably don't want the outside world accessing it! Thus, SSN would be `private`. Similar for any method that, say, validates the SSN within the class.

By default, if you don't use the `public` or `private` keywords, attributes and methods are considered `public`. This means that you can access `exteriorColor` by the following code:

```
System.out.println(MyCar.exteriorColor);
```

Since we created getters and setters to allow us to interact with the `exteriorColor` attribute, allowing direct access to the attributes within the class is probably not a good idea. Here's how we can rectify that situation using the `private` keyword:

```
private String exteriorColor;
private int numberOfCylinders;
```

At this point, the outside world can ONLY inquire or update the exterior color and/or the number of cylinders via their associated getters and setters.

Besides the keywords `public` and `private`, there is a third keyword: `protected`. We'll talk about that keyword later on.

Now, suppose you instantiate two objects from the class `Car`:

```
Car MyCar1 = new Car("PurplePassion",4);
Car MyCar2 = new Car("VomitYellow",8);
```

Be aware that the attributes associated with `MyCar1` do not affect those in `MyCar2`. That is, both objects are completely distinct.

However, suppose you want to keep track of the number of `Car` objects that have been instantiated (two in the example above). You can degrade yourself by keeping track on a piece of paper, but that's not going to cut it in the cut-throat world of object-oriented programming. A more appropriate way is to create a `static` variable in the `Car` class and update it within the constructor. A *static variable* (also known as a *class variable*) is shared across all instantiated objects from the same class. For example, if we add the following code to our `Car` class...

```
public static int objectCount = 0;
```

...we can go ahead and instantiate two cars as well as print out the value of `objectCount`...

```
Car MyCar1 = new Car("PurplePassion",4);
System.out.println("Number of Objects = " + MyCar1.objectCount);

Car MyCar2 = new Car("YellowVomit",8);
System.out.println("Number of Objects = " + MyCar2.objectCount);
```

Here's the output:

```
Number of Objects = 1
Number of Objects = 2
```

The keyword `static` on an *attribute* indicates that the variable is a *static* (or *class*) *variable.*

You can also have *static methods* within a class. This indicates that you do NOT have to instantiate the class in order to execute the method. For example,

```
class NumberInfo {

 public static double PI = 3.1415;

 public static double SquareIt(double pNum) {
  return(pNum*pNum);
 }

}

System.out.println("Cheap PI = " + NumberInfo.PI);
System.out.println("Square of 5 = " + NumberInfo.SquareIt(5));
```

The results are:

```
Cheap PI = 3.1415
Square of 5 = 25.0
```

Now, suppose your project requires that you not only keep track of car-specific information, but truck-specific information as well. You can probably see that both cars and trucks have the attributes exterior color and number of cylinders in common. While you can blindly create a `Car` class as well as a `Truck` class, does this make sense especially in light of the fact that several attributes and methods will overlap? No, it doesn't. If, instead, you create

a `Vehicle` class containing the common attributes and methods, you can then use the object-oriented concept of a *subclass* to create your `Car` and `Truck` classes. For example, let's create our `Vehicle` class:

```
class Vehicle {

 private String exteriorColor;
 private int numberOfCylinders;

 //Our constructor is below
 public Vehicle(String pExtClr,int pNumCyl) {
  exteriorColor = pExtClr;
  numberOfCylinders = pNumCyl;
 }

 //Getters
 public String getExteriorColor() {
  return(exteriorColor);
 }

 public int getNumberOfCylinders() {
  return(numberOfCylinders);
 }

 //Setters
 public void setExteriorColor(String pExtClr) {
  exteriorColor = pExtClr;
 }

 public void setNumberOfCylinders(int pNumCyl) {
  numberOfCylinders = pNumCyl;
 }

}
```

Next, let's create a `Car` class that extends the functionality of the `Vehicle` class. Take note that I've added a car-specific attribute: `ipodCharger` which is true if the car has a built-in iPod charger and false if not. Note that I've included both a getter and setter for this attribute within the `Car` class below:

```
class Car extends Vehicle {

 private boolean ipodCharger;

 public Car(String pExtClr,int pNumCyl,boolean pIPC) {
  super(pExtClr,pNumCyl);
  ipodCharger = pIPC;
 }

 public boolean getIpodCharger() {
  return(ipodCharger);
 }

 public void setIpodCharger(boolean pIPC) {
  ipodCharger = pIPC;
 }

}
```

As you see above, you use the `extends` keyword to indicate that the `Car` class will contain everything in the `Vehicle` class as well as additional stuff coded specifically in the `Car` class. Here's similar code for the `Truck` class with its truck-specific getter and setter for Gross Vehicular Weight:

```
class Truck extends Vehicle {

 private double grossVehicularWeight;

 public Truck(String pExtClr,int pNumCyl,double pGVW) {
  super(pExtClr,pNumCyl);
  grossVehicularWeight = pGVW;
 }

 public Double getGrossVehicularWeight() {
  return(grossVehicularWeight);
 }

 public void setGrossVehicularWeight(double pGVW) {
  grossVehicularWeight = pGVW;
 }

}
```

If you look at the constructors for both the `Car` and `Truck` classes, you'll see the following line of code:

```
super(pExtClr,pNumCyl);
```

This indicates that the constructor in the *superclass* - in this case, the `Vehicle` class - should be called from within the `Car` constructor and `Truck` constructor. This allows the variables `exteriorColor` and `numberOfCylinders` to be initialized since they appear within the `Vehicle` class and not the `Car` or `Truck` classes. If you did not include this line of code, both of these variables would be initialized to their default values (`0` for numbers and a single lonely blank for strings).

Subclassing is not an esoteric topic. The concept of extending a class is used quite a bit as we shall see later in this chapter.

Now, suppose you're not happy with the way the superclass's `getExteriorColor` method works. Are you stuck with it? Hell no! You can override this and other methods appearing in the superclass (`Vehicle`, in this example) and place the replacement code in your subclass (`Car` or `Truck`, in the examples below). For example, let's override the `getExteriorColor` method by placing the replacement code in our `Car` and `Truck` classes:

```
class Car extends Vehicle {

 private boolean ipodCharger;

 public Car(String pExtClr,int pNumCyl,boolean pIPC) {
  super(pExtClr,pNumCyl);
  ipodCharger = pIPC;
 }

 public boolean getIpodCharger() {
  return(ipodCharger);
 }

 public void getIpodCharger(boolean pIPC) {
  ipodCharger = pIPC;
 }
```

```
 //Override the getExteriorColor method appearing
 // in the Vehicle class with my own method.
 @Override
 public String getExteriorColor() {
  return("The exterior color for this CAR is " + super.getExteriorColor());
 }

}

class Truck extends Vehicle {

 private double grossVehicularWeight;

 public Truck(String pExtClr,int pNumCyl,double pGVW) {
  super(pExtClr,pNumCyl);
  grossVehicularWeight = pGVW;
 }

 public Double getGrossVehicularWeight() {
  return(grossVehicularWeight);
 }

 public void getGrossVehicularWeight(double pGVW) {
  grossVehicularWeight = pGVW;
 }

 //Override the getExteriorColor method appearing
 // in the Vehicle class with my own method.
 @Override
 public String getExteriorColor() {
  return("The exterior color for this TRUCK is " + super.getExteriorColor());
 }

}
```

Notice that in both the `Car` and `Truck` classes, we have overridden the `getExteriorColor` method with each class's own method. The keyword `@Override` is called an *annotation* and is used by the Java compiler to indicate that the current method is intended to override, or replace, the superclass's version of the method. The `@Override` annotation is not really needed and you can program without it, but its function is to tell the compiler to ensure that the superclass's method you're intending to override actually exists. That is, if you do NOT use the `@Override` annotation and you misspell your method in the subclass, you won't receive a compiler error because Java thinks you're creating a new method. But, your program won't work as expected since you haven't actually overridden the method you intended to. `@Override` will give you an error message like the one below:

```
[smithbob@lnxserver ~]$ javac jpgm5.java
jpgm5.java:79: method does not override or implement a method from a supertype
 @Override
    ^
1 error
```

Interface

As shown in the examples above, a class can be extended by a single base class. Java only allows for single inheritance and **does not allow multiple inheritance**. That is, you can't name more than one class after the `extends` keyword. To work around this debacle, interfaces were created. Java allows you to define an *interface* as a set of method signatures. By signatures, we mean that you don't, in fact, create the code within the method, but just define the method name, parameters, return type, etc. This is similar to function prototypes in C.

One author refers to an interface as *a scaled down mechanism to achieve multiple inheritance*. That is, a class can inherit from one or more interfaces (as well as a single base class, if needed). The class that inherits from an interface is responsible for defining the methods within that interface. Please re-read that last sentence!

Another author states that *an interface specifies what a class must do, but not how to do it*.

Some authors state that an interface is a *contract between two pieces of code*. That is, once a class inherits from an interface, that class is *guaranteed* to implement the methods of the interface (i.e., the program won't compile otherwise).

Other authors say that *coding to an interface, rather than an implementation, makes your software easier to extend*.

Still other authors say that an interface describes *behavioral characteristics or abilities* that can be *applied to* classes regardless of the class hierarchy. They say that classes, on the other hand, are responsible for *actions*. Personally, I prefer this description of interfaces rather than the others.

Now, to implement an interface, the programmer is responsible for implementing the classes defined in any pre-existing interface being using. On the other hand, if the programmer creates an interface, the programmer is responsible for coding the methods within it. Here's the syntax to define an interface:

```
access-modifier interface interface-name {

 //Define your attributes
 data-type var-name-1 = value-1;
 data-type var-name-2 = value-2;
 ...

 //Define your methods
 return-data-type method-name-1(parameter-list-1);
 return-data-type method-name-2(parameter-list-2);
 ...

}
```

where *access-modifier* can be `public` (which allows other packages to use the interface), or left off completely (which only allows the code within the package to access it). The remaining code should be apparent.

Note that some authors name their interfaces starting with a capital letter `I` followed by the rest of the interface name. This is not set in stone, so follow your heart, poppets! For example, `IColor`, `ISize`, `IRadio`, etc. are all names of interfaces and don't, in any way, reflect the programmer's own self-absorption.

For example, let's create an interface which defines the characteristics or behaviors of a car radio:

```
interface IRadio {
 public bool bOn=false;
 public String sBand="AM";
 public float fHertz=1060;

 public void turn_on_off(bool bOn);
 public void change_station(float fHertz);
 public void change_band(String sBand);
}
```

Any variables defined in an interface are implicitly `public` as well as `final` and `static`. All variables must be initialized within an interface's definition.

Now, to use your interface, you add the `implements` keyword followed by the name of the interface after the class name, or after the `extends` clause:

```
class Car extends Vehicle implements IRadio {
 ...fill in the methods here...
}
```

It's within your class where you implement the methods defined in the interface itself. As mentioned above, you can specify more than one interface, as shown below:

```
class Car extends Vehicle implements IRadio, ICoffeeMachine, IAspirinDispenser
{
 ...fill in the methods here...
}
```

Abstract

As described above, the programmer is responsible for implementing all of the classes defined within the interface. While using interfaces is a perfectly reasonable way to go, all you may want to do is create a method in your class that the final programmer is responsible for implementing. You may define the method's signature, but due to not knowing how the target programmer will use the method, you may want to code the method yourself. This is where the `abstract` keyword comes in. By defining a method of your class as `abstract`, you're saying to the target programmer (that is, the programmer using your class): *I don't know in what context you will be using this method; all I know is that the rest of my code needs it and you have to implement this method yourself. So, get to work, buddy!*

Note that this differs from using `@Override` annotation shown previously. The `@Override` annotation indicates that **your** *fully-implemented-method* will replace **their** *fully-implemented-method*.

Note that if you have one abstract method in your class, the class itself must be defined as abstract. This does not mean that everything within your class must be implemented by the target programmer, and your class can contain fully implemented code within its stone fortress walls.

Note that *an abstract class cannot be instantiated*. Instead, you must place the name of the abstract class to the right of the `extends` keyword.

For example, in the code above, we could have defined the `Vehicle` class as abstract as well as its `getExteriorColor()` method.

```
abstract class Vehicle {

 private String exteriorColor;
 private int numberOfCylinders;

 //Our constructor is below
 public Vehicle(String pExtClr,int pNumCyl) {
  exteriorColor = pExtClr;
  numberOfCylinders = pNumCyl;
 }

 //Getters
 abstract public String getExteriorColor();

 public int getNumberOfCylinders() {
  return(numberOfCylinders);
 }

 //Setters
 public void setExteriorColor(String pExtClr) {
  exteriorColor = pExtClr;
 }
```

```
 public void setNumberOfCylinders(int pNumCyl) {
  numberOfCylinders = pNumCyl;
 }

}
```

Now, when extending the `Car` class with the `Vehicle` class, you must code the `getExteriorColor()` method yourself. Here's the full code:

```
import java.util.*;

//Here`s the abstract Vehicle class.
abstract class Vehicle {

 private String exteriorColor;
 private int numberOfCylinders;

 //Our constructor is below
 public Vehicle(String pExtClr,int pNumCyl) {
  exteriorColor = pExtClr;
  numberOfCylinders = pNumCyl;
 }

 //Getters
 abstract public String getExteriorColor();

 public int getNumberOfCylinders() {
  return(numberOfCylinders);
 }

 public String getColor() {
  return(exteriorColor);
 }

 //Setters
 public void setExteriorColor(String pExtClr) {
  exteriorColor = pExtClr;
 }

 public void setNumberOfCylinders(int pNumCyl) {
  numberOfCylinders = pNumCyl;
 }

}

//Here`s the Car class.
class Car extends Vehicle {

 private boolean ipodCharger;

 public Car(String pExtClr,int pNumCyl,boolean pIPC) {
  super(pExtClr,pNumCyl);
  ipodCharger = pIPC;
 }

 public boolean getIpodCharger() {
  return(ipodCharger);
 }

 public void getIpodCharger(boolean pIPC) {
```

```
  ipodCharger = pIPC;
 }

 //Create the code for the abstract method getExteriorColor.
 public String getExteriorColor() {
  return("The exterior color for this CAR is " + getColor());
 }

}

//Here`s the Truck class.
class Truck extends Vehicle {

 private double grossVehicularWeight;

 public Truck(String pExtClr,int pNumCyl,double pGVW) {
  super(pExtClr,pNumCyl);
  grossVehicularWeight = pGVW;
 }

 public Double getGrossVehicularWeight() {
  return(grossVehicularWeight);
 }

 public void getGrossVehicularWeight(double pGVW) {
  grossVehicularWeight = pGVW;
 }

 //Create the code for the abstract method getExteriorColor.
 public String getExteriorColor() {
  return("The exterior color for this TRUCK is " + getColor());
 }

}

class jpgm7 {

 public static void main(String args[]) {

  Car MyCar = new Car("PurplePassion",5,true);
  Truck MyTruck = new Truck("BisonBrown",5,56.7643);

  //Print out the exterior color of the car and truck.
  System.out.println(MyCar.getExteriorColor());
  System.out.println(MyTruck.getExteriorColor());

 }

}
```

After compiling and executing this code, the results are:

```
[smithbob@lnxserver ~]$ java jpgm7
The exterior color for this CAR is PurplePassion
The exterior color for this TRUCK is BisonBrown
```

Variable-Length Arguments

In the methods shown above, there was exactly one parameter for each desired parameter. This makes complete sense! However, Java allows you to define a variable-length argument using the ellipsis notation. An ellipsis is just three periods following the data type but before the name of the parameter. For example, here's a method that sums up a series of numbers passed in via the variable-length argument:

```
public double SumTotal(double... nums) {

 double tot=0.0;

 for(int i=0;i<nums.length;i++) {
  tot += nums[i];
 }

 return(tot);
}
```

As you see above, an ellipsis follows the `double` data type. As shown above, you can access the individual elements of the `nums` using the array bracket notation, or you can use the following nifty notation instead:

```
public double SumTotal(double... nums) {
 double tot=0.0;

 for(double anum : nums) {
  tot += anum;
 }

 return(tot);
}
```

To use this method, provide a comma-delimited list of values or variable to the method:

```
double GrandTotal_Qtr1 = SumTotal(Qtr1,Qtr2,Qtr3);
```

Note that if you have more than one argument on your method, the ellipsis must appear as the last parameter in the list! You cannot include it in the middle or the beginning. Naturally, this doesn't matter if you only have a single "parameter", as shown in the example above.

Running Java Test Programs

While you're attempting to learn Java, you may want to create a few test programs and then compile them using the Java compiler (`javac`). To create a quick Java test program, open up a text editor (such as the `vi` Editor), type in some spiffy Java code, compile the program and then run it. Let's create a test program, called `testpgm.java`, which just simply prints out the word `BOINK!`.

Open the new file `testpgm.java` in the `vi` Editor: `vi` **`testpgm`**`.java`. Insert the following text into the file:

```
import java.util.*;
import java.lang.*;

public class testpgm {

 public static void main(String[] args) throws Exception {
  System.out.println("BOINK!");
 }

}
```

Ensure that the name of the class matches the name of your file (**testpgm**, here)! Save and exit out of the `vi` Editor (`:wq`).

Next, let's compile the program at the Linux command line:

```
[smithbob@lnxserver ~]$ javac testpgm.java
```

Assuming there are no error messages, let's run the program:

```
[smithbob@lnxserver ~]$ java testpgm
```

If all went well, you should see the following:

```
BOINK!
```

Huzzah!!

Exploring Useful Java Classes

In this section, we'll explore several useful Java classes such as the `String` class, `Math` class, and so on. Although not every method in every class will be discussed, we do present enough information to justify why you should know about a particular class, show you basic examples to get you started, and point you in the right direction if further information is required.

If you would like to peruse a list of all of the classes available in Java 8, navigate your browser to…

`http://docs.oracle.com/javase/8/docs/api/allclasses-noframe.html`

…but be aware that there are well over `4000` classes listed, so you might want to make yourself a nice cup o' tea.

We'll explore the following classes:

- Text-related Classes
  - `String` - this class contains methods used to manipulate strings.
  - `RegExp` - this class contains methods used to work with regular expressions.
- Mathematics-related Classes
  - `Math` - this class contains mathematical constants and methods used to work with numbers.
  - `Integer` - this class contains methods used to work with integers.
  - `Big Decimal` - this class contains methods used to work with integral values that exceed the minimum and maximum permitted values of the data types `int` and `long`.
  - `Random` - this class contains methods used to produce random numbers.
- Collections-Related Classes
  - `Arrays` - this class contains methods useful when working with arrays.
  - `ArrayList` - this class contains methods useful when you need to work with an unordered list.
  - `HashMap` - this class contains methods useful when you need to work with a series of key/value pairs.
- Date- and Time-Related Classes
  - `Date` - this class contains methods useful when working with dates and times.
  - `SimpleDateFormat` - this class contains methods useful when working with and formatting dates and times.

These classes can help you write useful and efficient code as well as prevent you from reinventing the wheel, something we've all done at least once in the past...heavy sigh...

Text-Related Classes

In this section, we explore the `String` and `RegExp` classes.

You can access these two classes by importing the Java packages `java.lang` and `java.util.regex` by placing the following two lines of code at the top of your Java program:

```
import java.lang.*;
import java.util.regex.*;
```

Note that we've worked with creating text strings using the `String` class in a limited way above:

```
String sBand="AM";
```

...or...

```
String exteriorColor;
```

In both cases, `sBand` and `exteriorColor` are now `String` objects and have access to the myriad of methods available to that particular class.

Another way to instantiate a `String` object is to use one of the many constructors (recall we talked about constructors earlier), although you may only ever use the following one to create an empty `String`:

```
String sBand = new String();
```

But, you're more likely to code this instead:

```
String sBand = "";
```

Now, the following methods of the `String` class are useful to know:

- ☐ `charAt(`*`index`*`)` – this method returns the character located at *index* in your `String`.
- ☐ `concat(`*`addString`*`)` – this method concatenates your `String` and *`addString`* together, similar to the code `String + addString`.
- ☐ `indexOf(`*`char`*`)` – this method returns the position of *`char`* within your `String`.
- ☐ `indexOf(`*`search`*`,`*`index`*`)` – this method, similar to the above, returns the position of the search term *`search`* starting at *`index`* within your `String`.
- ☐ `length()` – this method returns the length of your `String` as an `int`.
- ☐ `split(`*`regexp`*`)` – this method splits your `String` into an array of `String`s using the provided regular expression, *`regexp`*.
- ☐ `substring(`*`start`*`,`*`end`*`)` – this method returns a substring of your `String` starting at *`start`* (the offset of the first character) and ending at *`end`* (the offset one past the last character). A variation of this method, without the *`end`* argument, returns the substring from *`start`* to the end of the entire string.
- ☐ `toLowerCase()` – this method converts your `String` to lowercase.
- ☐ `toUpperCase()` – this method converts your `String` to UPPERCASE.
- ☐ `trim()` – this method removes both leading and trailing whitespace characters from your `String`.

For example, given the following `String`:

```
String sFiller = "The quick brown fox jumps over the lazy dog.";
```

Let’s create upper- and lowercase versions of it:

```
String sFiller_Upper = sFiller.toUpperCase();
String sFiller_Lower = sFiller.toLowerCase();
```

And the results, as you might expect, are:

```
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
the quick brown fox jumps over the lazy dog.
```

Now, let's substring `sFiller` so that we end just before the word **dog**. Now, create a string to hold our search term:

```
String sSearchTerm = "dog.";
```

Next, find the location of the start of this search term within `sFiller`:

```
int iSearchTermLocation = sFiller.indexOf(sSearchTerm);
```

Finally, take the substring from the beginning of `sFiller` (starting at zero) and ending at `iSearchTerm Location`:

```
String sFillerNoDog = sFiller.substring(0,iSearchTermLocation);
```

The results are predictable, but be aware that there's a blank space at the end of this string:

```
The quick brown fox jumps over the lazy
```

Next, let's concatenate the word `moose` to our new `String`, `sFillerNoDog`:

```
String sLargeAnimal = "moose.";
String sFillerWithMoose = sFillerNoDog.concat(sLargeAnimal);
```

And the results are:

```
The quick brown fox jumps over the lazy moose.
```

One nice method is the `split()` method which breaks apart a `String` using the provided regular expression as the argument. The result of `split()` is an array of `String`s containing substrings of the original `String`. For example, let's break apart `sFiller` at the blanks by creating a `String` to hold the desired regular expression. Note that we have to escape the first backslash:

```
String sRegExp = "\\s+";
```

Next, let's use the `split()` method to break apart `sFiller`:

```
String[] sFillerSplits = sFiller.split(sRegExp);
```

And, let's display each piece of the array `sFillerSplits`:

```
for(String s : sFillerSplits) {
 System.out.println(s);
}
```

And the results are as follows:

```
The
quick
brown
fox
jumps
over
the
lazy
```

```
dog.
```

You can use more complicated regular expressions than the one shown above by using the classes and methods provided in the `java.util.regex` package.

In order to use regular expressions in Java, you first compile your regular expression and then use the compiled regular expression for matches, replacements, and so on.

Now, by *compile* we don't mean you need to create a separate Java program, but just use the `compile()` method of the `Pattern` class along with your desired regular expression. You normally do this if you'll be using a particular regular expression many times, say, by trying to parse millions of addresses stored within individual text strings.

For example, given the following address,

```
123 NORTH MAIN STREET SOUTH SUITE A123
```

Let's use regular expressions to separate out each address component into individual pieces using the following regular expression:

```
String sRE = "^(\\d+)
+(NORTH|NORHT|NRTH|SOUTH|SOUHT|SUOTH|EAST|EASST|WEST|WESST{1}) +(\\w+)
+(ST|STREET|STREE|STEET|STREEET|STRET{1})
+(NORTH|NORHT|NRTH|SOUTH|SOUHT|SUOTH|EAST|EASST|WEST|WESST{1})
+(SUITE|SUTIE|SUIT|SUI|STE{1}) +(\\w+) *$";
```

Now, this regular expression makes use of *alternation* – the vertical bars used to represent an OR condition – in order to capture misspellings. Please see *Chapter 11 – Regular Expressions* for more on regular expressions. For example, the address below will also be captured by the regular expression above because we took into account possible misspellings:

```
123 NRTH MAIN ST SUOTH STE A123
```

We also make use of capturing groups – the left and right parentheses – in order to capture the individual pieces within the completely matched regular expression. Each capturing group is numbered starting from the left-most position in the regular expression. The first capturing group is numbered `1`, the second is `2`, and so on. This will come in handy when we use the `groups()` method in order to work with each individual piece.

Now that we've created our regular expression, let's compile it:

```
Pattern oRE = Pattern.compile(sRE);
```

Next, let's search for matches using the `matcher()` method to return a `Matcher` object:

```
Matcher oMATCH = oRE.matcher(sAddress);
```

At this point, we can ask if there are any matches using the `matches()` method of the `Matcher` object `oMATCH`:

```
if (oMATCH.matches()) {
```

And, if there are matches, we can pull the individual pieces using the `groups()` method of the `Matcher` object `oMATCH` providing the capturing group number (equivalent to `\1`, `\2`, etc.):

```
if (oMATCH.matches()) {

 String sHOUSE_NUMBER = oMATCH.group(1);
 System.out.println(sHOUSE_NUMBER);

 String sDIR_PRE = oMATCH.group(2);
 System.out.println(sDIR_PRE);
```

```
 String sSTREET_NAME = oMATCH.group(3);
 System.out.println(sSTREET_NAME);

 String sSTREET_TYPE = oMATCH.group(4);
 System.out.println(sSTREET_TYPE);

 String sDIR_POST = oMATCH.group(5);
 System.out.println(sDIR_POST);

 String sSUITE_TYPE = oMATCH.group(6);
 System.out.println(sSUITE_TYPE);

 String sSUITE_NBR = oMATCH.group(7);
 System.out.println(sSUITE_NBR);

}
```

Here are the results:

```
123
NRTH
MAIN
ST
SUOTH
STE
A123
```

Now, regular expressions don't need be as complex as the one shown above. If you just want to search for a particular pattern, say a series of three numbers occurring multiple times within a text string, you can use the `find()` method of the `Matcher` class to return each occurrence appearing within the matched string. For example, let's create a `String` with a series of three numbers:

```
String sCodeNumbers = "123 456 789 012";
```

Next, let's create the regular expression to search for sets of three numbers:

```
String sRE = "\\d{3}";
```

Next, let's compile it and create the `Matcher` object:

```
Pattern oRE = Pattern.compile(sRE);
Matcher oMATCH = oRE.matcher(sCodeNumbers);
```

And, finally, let's loop through all of the matches:

```
while (oMATCH.find()) {
 System.out.println("Code Number=" + oMATCH.group());
}
```

And here are the results:

```
Code Number=123
Code Number=456
Code Number=789
Code Number=012
```

Please see the documentation for more on regular expressions as well as the `Pattern` and `Matcher` classes.

Mathematics-Related Classes

In this section, we explore the `Math`, `Random`, `Integer` and `BigInteger` classes.

You can access these classes by importing the Java packages `java.util`, `java.math` and `java.lang` by placing the following lines of code at the top of your Java program:

```
import java.lang.*;
import java.math.*;
import java.util.*;
```

The `Math` class provides basic mathematical constants (such as `E` and `PI`) as well as a plethora of mathematical functions (such as `abs()`, `cos()`, and so on). For example, if you'd like to use the value for pi in your Java program, you can code something like this:

```
double myPI = Math.PI;
```

But, you can just as easily refer to `Math.PI` within your program instead of creating a separate variable.

Now, the great thing about the `Math` class is that all of the methods are `static` meaning that you don't have to instantiate a `Math` object in order to use a particular method. For example, let's take the absolute value of a number:

```
double myDouble = -2.457584743638;
System.out.println(Math.abs(myDouble));
```

You're already familiar with many of the methods that make up the `Math` class from the list of functions available in ImpalaSQL, here's an abbreviated list of methods for you to peruse:

- `abs()` – this method returns the absolute value.
- `acos()` – this method returns the arc cosine.
- `asin()` – this method returns the arc sine.
- `atan()` – this method returns the arc tangent.
- `atan2()` – this method returns the arc tangent.
- `ceil()` – this method returns the ceiling.
- `cos()` – this method returns the cosine.
- `cosh()` – this method returns the hyperbolic cosine.
- `exp()` – this method returns powers of e (=`2.7182818`...).
- `floor()` – this method returns the floor.
- `hypot()` – this method computes the hypotenuse.
- `log()` – this method returns the log (base `e`).
- `log10()` – this method returns the log (base `10`).
- `max()` – this method returns the maximum.
- `min()` – this method returns the minimum.
- `pow()` – this method raises a number to a power.
- `random()` – this method returns a pseudo-random number.
- `round()` – this method rounds a number to a specified number of decimal places.
- `sin()` – this method returns the sine.
- `sinh()` – this method returns the hyperbolic sine.
- `sqrt()` – this method returns the square root.
- `tan()` – this method returns the tangent.
- `tanh()` – this method returns the hyperbolic tangent.
- `toDegrees()` – this method converts radians to degrees.
- `toRadians()` – this method converts degrees to radians.

Take note of the method `random()`. This method returns a pseudo-random number, as a `Double`, between zero (inclusive) and one (exclusive). For example, let's produce several random numbers:

```
double myRandNbr;
for(int i=0;i<10;i++) {
 System.out.println(Math.random());
}
```

And here are the results:

```
0.9937116662147903
0.8832714457691465
0.021470912780462093
0.06808048975506253
0.2272317374013757
0.4853108586915391
0.003877548002589437
0.04306726829210694
0.13320416155649084
0.9509692890645647
```

Note that each time you run the code you'll receive different random numbers.

Now, if you'd like more control over your random numbers, you can use the `Random` class instead of the `random()` method of the `Math` class. There are two constructors to the `Random` class:

- ☐ `Random()` – this constructor constructs a random generator with an initial state that is unlikely to be duplicated by a subsequent instantiation.
- ☐ `Random(`*`seed`*`)` – this constructor creates a random generator using *`seed`* as the initial state. Note that *`seed`* is a `long` data type.

Now, the `Random` class contains several methods to generate random numbers of a specific data type. For example, you can use the `nextDouble()` method to generate random numbers between zero (inclusive) and one (exclusive) as a `double` data type. On the other hand, if you just want a series of random integers between zero (inclusive) and some number `n` (exclusive), you can use the `nextInt(n)` method instead.

For example, let's generate a series of random numbers using these two methods:

```
Random oRand = new Random();
int iRandomInteger;
double dRandomDouble;

for(int i=0; i<5; i++) {
 System.out.println(oRand.nextDouble());
 System.out.println(oRand.nextInt(100));
}
```

And the results are:

```
0.7262704595568443
96
0.7307144140023264
22
0.8026506976431572
98
0.8235707998746877
46
0.23907124476393293
2
```

Now, let's discuss the `Integer` class. Occasionally, you'll want to convert an `int` value into a `String` and attempts to do the following will **fail**:

```
String sInteger = iInteger.toString(); //FAIL!!
```

In order to convert an `int` to a `String`, you must make use of the `toString()` method of the `Integer` class, like so:

```
int iInteger = 3;
String sInteger = Integer.toString(iInteger);
```

Now, there are many methods available in the `Integer` class and we discuss some of them below, but when perusing the Java documentation, take note that some of the methods are `static` and others are not. Recall that `static` methods don't need an instantiated class to be used.

For example, you can compare two `ints` using the `compare()` method to determine if the values are greater than, less than, or equal to each other.

```
int iInteger1 = 3;
int iInteger2 = 5;
int iReturnValue = Integer.compare(iInteger1,iInteger2);
System.out.println(Integer.toString(iReturnValue));
```

Now, if `iInteger1` is less than `iInteger2`, a negative number will be returned (usually a `-1`). If `iInteger1` is greater than `iInteger2`, a positive number will be returned (usually a `+1`). If both numbers are the same, then a zero will be returned. In the example above, `-1` was returned.

Since some unknown negative or positive value will be returned by the `compare()` method, we can use the static method `signum()` to convert the values returned by the `compare()` method to `-1`, `+1` and `0`:

```
int iReturnValue = Integer.signum(Integer.compare(iInteger1,iInteger2));
```

Another nice static method of the `Integer` class is the `bitCount()` method which returns the number of `1` bits within a given `int`. This is also known as the *population count*. For example, the integer `12893` is represented by the binary number `11001001011101`. Counting the number of `1`s in the binary number yields `8`, the population count. Let's see that in code:

```
int iInteger = 12893; /* 11001001011101 = 8 one bits in total */
int iPopCnt = Integer.bitCount(iInteger);
System.out.println(Integer.toString(iPopCnt));
```

The result is, of course, `8`.

Although we only discussed the `Integer` class, please see the documentation for similar classes such as `Boolean`, `Byte`, `Double`, `Float`, `Long`, `Number` and `Short`.

Next, let's talk about the `BigInteger` class. Recall that the range for an integer is between `-2,147,483,648` and `2,147,483,647`. Let's see what happens if we attempt to set an `int` to `214748364`**`8`**:

```
int iBigNumber = 2147483648;

programname.java:19: error: integer number too large: 2147483648
        int iBigNumber = 2147483648;
                         ^
1 error
```

Now, we could use a `long` data type, but that is limited to a maximum value of `9,223,372,036,854,775,807` and, hence, has the same potential problem.

Now, if you need arbitrary precision integer numbers, you can use the `BigInteger` class. For example, let's use one of the many constructors to take a `String` containing these two very large numbers and instantiate two `BigInteger`s:

```
BigInteger oBI_ExceedsINT = new BigInteger("2147483648");
BigInteger oBI_ExceedsLONG = new BigInteger("9223372036854775808");
```

Next, let's add these two numbers together using the `add()` method and print out the result:

```
BigInteger oBI_TOTAL = oBI_ExceedsINT.add(oBI_ExceedsLONG);
System.out.println(oBI_TOTAL.toString());
```

And, the results are: `9223372039002259456`.

If need be, we can create a `double` to represent our sum total:

```
double dBI_TOTAL = oBI_TOTAL.doubleValue();
```

Besides `BigInteger`, please see `BigDecimal` in the Java documentation.

Finally, if you need your mathematical constants and functions to return exactly the same values across all platforms, please see the `StrictMath` class in the `java.lang` package.

Collections-Related Classes

In this section, we explore the `Arrays`, `ArrayList` and `Hashmap` classes.

You can access these classes by importing the Java package `java.util`:

```
import java.util.*;
```

Recall that you can create an array by using the left and right brackets(`[]`) after the data type. For example, to initialize an array of integers from `1` to `10`, you can code something like this:

```
int[] aiNumbers = {1,2,3,4,5,6,7,8,9,10}; // Take note of the curly braces‼
```

But, the `Arrays` class allows you to work with arrays more easily. For example, suppose you want to perform a binary search of the `aiNumbers` array above. The `Arrays` class contains several overrides of the `binarySearch()` method which returns the zero-based index of the desired value. For example, let's retrieve the index of the number `9` in the `aiNumbers` array:

```
int iIndex = Arrays.binarySearch(aiNumbers,9);
```

Since arrays are zero-based, the resulting index is `8`.

Unfortunately, `binarySearch()` requires that the array is sorted. If your array isn't sorted, you can use the `sort()` method prior to using the `binarySearch()` method:

```
Arrays.sort(aiNumbers);
```

Note that the return type for the `sort()` method is `void`, so you can't use it in the argument to the `binarySearch()` method.

If you'd like to copy a portion of your array out to another array, you can use the `copyOfRange()` method. For example, let's create a new array made up of the fourth through the seventh array elements (the numbers `4`, `5`, `6`, and `7`):

```
int[] aiNumbersSubset = Arrays.copyOfRange(aiNumbers,3,7);
```

Again, since arrays are zero-based, the beginning of the range (the second argument) is set to `3`. Also, note that the end of the range (the third argument) is one more than what we need, `7` in this case.

Now, if you'd like to work with a list instead of an array, you can use the `ArrayList` class. According to the Java documentation: An `ArrayList` is an implementation of `List`, backed by an array. All operations including adding, removing, and replacing elements are supported. A `List` is a collection which maintains an ordering for its elements. Every element in the `List` has an index. Each element can thus be accessed by its index, with the first index being zero. Normally, `List`s allow duplicate elements, as compared to `Set`s, where elements have to be unique.

For example, let's create an `ArrayList` to hold the names of several US states:

```
ArrayList<String> oAL_STATES = new ArrayList<String>();
```

Take note of the word `String` within angled brackets (`<>`) above. This indicates to `ArrayList` which data type it should enforce when adding elements to the list. If you don't include this bracketed syntax, called *generics*, Java will barf at you with the following error message:

```
Note: programname.java uses unchecked or unsafe operations. You suck, dude!
```

Next, let's add values to our `ArrayList`:

```
oAL_STATES.add("Ohio");
oAL_STATES.add("Wyoming");
oAL_STATES.add("California");
oAL_STATES.add("Alabama");
```

Now, let's print out the values in our `ArrayList`:

```
for(String sThisState : oAL_STATES) {
 System.out.println(sThisState);
}
```

The results are predictable:

```
Ohio
Wyoming
California
Alabama
```

Another way to produce the same result is by using the `get()` and `size()` methods to pull each individual `ArrayList` element:

```
for(int i=0; i<oAL_STATES.size(); i++) {
 String sThisState = oAL_STATES.get(i);
 System.out.println(sThisState);
}
```

Besides the `get()` method to retrieve an item, you can use the `remove()` method to remove an item and the `set()` method to replace an item.

So far we've worked with arrays, the `Arrays` class and the `ArrayList` class. If you'd like to work with key/value pairs rather than just items in an array or list, you can use the `HashMap` class. According to the Java documentation: `HashMap` is an implementation of `Map`. All elements are permitted as keys or values, including null. Note that the iteration order for `HashMap` is non-deterministic. If you want deterministic iteration, use `LinkedHashMap`. A `Map` is a data structure consisting of a set of keys and values in which each key is mapped to

a single value. The class of the objects used as keys is declared when the `Map` is declared, as is the class of the corresponding values.

For example, let's create a `HashMap` to hold the four state names shown above along with the corresponding state capitals. For example, the capital of Wyoming is the W. Ha! That's a little joke! Ahem. The capital of Wyoming is Cheyenne.

```
HashMap<String,String> oHM_STATES = new HashMap<String,String>();
```

Note that in the code above we specified the keyword `String` twice in the generics separated by a comma. The first `String` indicates the data type of the **key** and the second `String` indicates the data type of the **value**. Since state and capital names are text, we specified both positions in the generic as `String`.

Next, let's add some key/value pairs to our `HashMap` using the `put()` method:

```
oHM_STATES.put("Ohio","Columbus");
oHM_STATES.put("Wyoming","Cheyenne");
oHM_STATES.put("California","Sacramento");
oHM_STATES.put("Alabama","Montgomery");
```

Note that the first argument indicates the **key** and the second indicates the **value**.

Next, let's iterate through all of the keys. You can do this by using the `keySet()` method:

```
for(String sThisState : oHM_STATES.keySet()) {
 System.out.println(sThisState);
}
```

Similarly, you can iterate through the values using the `values()` method:

```
for(String sThisCapital : oHM_STATES.values()) {
 System.out.println(sThisCapital);
}
```

Next, let's retrieve the state capital of `Wyoming`. We do this using the `get()` method:

```
String sWyoming_Capital = oHM_STATES.get("Wyoming");
```

And the result is, of course, `Cheyenne` (and not the `W`...*tee-hee!*).

Date- and Time-Related Classes

In this section, we explore the `SimpleDateFormat` and `Date` classes.

You can access these two classes by importing the Java packages `java.util` and `java.text`:

```
import java.util.*;
import java.text.*;
```

The `Date` class, despite its name, is used to hold times as well as dates and is accurate to the millisecond, which is a time saver. In order to initialize the `Date` object to the current date and time, just code:

```
Date oTodaysDateTime = new Date();
```

In order to see what date and time were returned, you can use the `toString()` method on `oTodaysDateTime`:

```
System.out.println("Today's Date and Time = " + oTodaysDateTime.toString());
```

Note that the results are printed in a specific format that looks like this:

```
Today's Date and Time = Thu Jul 10 09:48:19 EDT 2014
```

Now, this particular output format may not be to your liking, so you can use the `SimpleDateFormat` class to not only produce dates and times in a specific textual format, but parse a `String` containing date/time information to create a `Date` object.

For example, let's create a `String` from today's date in the format `MM/dd/yyyy`, or two-digit month, two-digit day and four-digit year all separated by forward slashes. First, instantiate a `SimpleDateFormat` object telling it which format will be used to output today's date:

```
SimpleDateFormat oSDF1 = new SimpleDateFormat("MM/dd/yyyy");
```

Next, let's retrieve today's date:

```
Date oTodaysDate = new Date();
```

Now, let's create a `String` from today's date. Note that since we specified `MM/dd/yyyy` when instantiating the `SimpleDateFormat` class, the resulting output will be in that specific format. Here, we're using the `format()` method supplying it the desired date, `oTodaysDate`:

```
String sDateInMMDDYYYYFmt = oSDF1.format(oTodaysDate);
System.out.println("Today's date is " + sDateInMMDDYYYYFmt);
```

And, the results are shown below:

```
Today's date is 07/10/2014
```

Now, the letters, `MM`, `dd`, `yyyy`, and so on are called the *Time Pattern Syntax*. Here are some of the more common patterns:

- ☐ `M` – month as a single digit (`1`=January...`12`=December)
- ☐ `MM` – month as two-digits (`01`=January...`12`=December)
- ☐ `MMM` – month as three-letters (Jan=January...Dec=December)
- ☐ `MMMM` – month as full text (January...December)
- ☐ `d` – day of month as a single digit (`1`...`31`)
- ☐ `dd` – day of month as two-digits (`01`...`31`)
- ☐ `yy` – two-digit year (`14` for `2014`)
- ☐ `yyyy` – four-digit year (`2014`)
- ☐ `H` – hours in 24-hour clock (`0` to `23`)
- ☐ `h` – hours in 12-hour clock (`1` to `12`)
- ☐ `m` – minutes (`0` to `59`)
- ☐ `s` – seconds (`0` to `59`)
- ☐ `a` – AM or PM indicator
- ☐ `EEE` – day of the week as three-letters (Mon=Monday...Sun=Sunday)
- ☐ `EEEE` – day of week as full text (Monday...Sunday)

For example, to specify today's date as `Thursday July 10, 2014 10:16:56 AM` we will use the pattern `"EEEE MMMM dd, yyyy @ hh:mm:ss a"`:

```
Date oTodaysDateTime = new Date();
SimpleDateFormat oSDF = new SimpleDateFormat("EEEE MMMM dd, yyyy hh:mm:ss a");
String oMyDate = oSDF.format(oTodaysDateTime);
System.out.println("Today's Date and Time = " + oMyDate);
```

And the results are shown below:

```
Today's Date and Time = Thursday July 10, 2014 10:16:56 AM
```

Next, let's go the other way round: let's produce a `Date` object from a `String` containing a specific date (not necessarily today's date).

First create a `String` to hold the desired date: `03/21/1962`:

```
String sMyDate="03/21/1962";
```

Next, instantiate `SimpleDateFormat` specifying the format `MM/dd/yyyy` which will be used to input the date `03/21/1962`:

```
SimpleDateFormat oSDF = new SimpleDateFormat("MM/dd/yyyy");
```

Create a `ParsePosition` object and sets its argument to zero. This object is used for debugging if there's ever a problem with your time pattern syntax. We won't discuss this object here, so please see the documentation for more.

```
ParsePosition oParsePosition = new ParsePosition(0);
```

Finally, we call the `parse()` method to read in our textual date and produce a `Date` object:

```
Date oMyDate = oSDF.parse(sMyDate,oParsePosition);
System.out.println("03/21/1962 as a Date Object = " + oMyDate.toString());
```

The results are shown below using Date's `toString()` method:

```
03/21/1962 as a Date Object = Wed Mar 21 00:00:00 EST 1962
```

Please see the documentation for `SimpleDateFormat`, `Date` as well as `ParsePosition` to learn more about these classes.

Chapter 40 – Creating User-Defined Functions (UDFs) for ImpalaSQL

With that very brief overview of Java complete, we can discuss user-defined functions in ImpalaSQL. Note that there are several types of user-defined functions you can create:

- ☐ **User-Defined Function** (UDF) – This type of function, similar to most of the functions available in ImpalaSQL, is executed for each row in a query and returns a single value. This function can be created using Java and can be accessed via ImpalaSQL.
- ☐ **User-Defined Aggregate Function** (UDA) – This type of function is similar to how the aggregate functions `SUM`, `AVG`, `MIN`, etc. operate in that data is summarized across one or more rows either with or without a `GROUP BY` Clause. This type of function can be created using Java, but is only accessible via HiveQL, not ImpalaSQL. In order to make this type of function available in ImpalaSQL, it would have to be programmed in C++…not an easy task for a mere mortal such as myself.
- ☐ **User-Defined Table Function** (UDT) – These functions, similar to analytic/windowing functions, are currently not supported by Impala.

In this chapter, we'll stick to creating a simple user-defined function (UDF) in Java for use with ImpalaSQL.

Creating a User-Defined Function (UDF)

When none of the functions available in ImpalaSQL meet your needs, you can always roll your own user-defined function (UDF) using Java. For the example used in this section, we create a user-defined function to shift an integer date, formatted in `yyyymm` format, by a certain number of months, either forward or backward in time. For example, given `202201`, shifting by `+2` months yields `202203`.; and, shifting by `-1` month yields `202112`. Both the date as well as the shift value are passed into the function as integers; the function itself returns an integer as well. The Java class we create is called `MonthIdShifter` and all of the Java code is placed in a file named `MonthIdShifter.java`.

In order to create a UDF, you must first import the appropriate UDF package at the top of your Java program:

```
import org.apache.hadoop.hive.ql.exec.UDF;
```

Next, you create your own public class which extends the UDF class:

```
public class your-class-name extends UDF {
```

Note: The class name must be the same as the name of the file which contains the code (with the exception of the `.java` extension).

Next, create a public method named `evaluate` which takes zero, one or more arguments passed in from the ImpalaSQL code. For example, the method `evaluate` below expects two arguments: the month identifier as an `Integer` in `yyyymm` format and the number of months to shift it by (positive or negative):

```
public int evaluate(Integer iMonthid,Integer iShiftValue) {
```

Take note that we define the return type to be an `int` because that data type contains a perfectly valid range for our output result.

Now, along with the UDF package, you must import any additional packages used throughout your Java code. In this example, we import the following packages:

```
import org.apache.hadoop.hive.ql.exec.UDF;
import java.util.*;
import java.text.SimpleDateFormat;
import org.apache.commons.lang3.StringUtils;
```

The `import` line `java.util.*` brings in, according to the Java docs website, *the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array)*. So, all good stuff.

The `import` line `java.text.SimpleDateFormat` allows you to easily format a date in the `yyyymm` format given a full-blown date.

Finally, the `import` line `org.apache.commons.lang3.StringUtils` brings in several string-related functions that are null safe. For example, we use the `substring()` function below to piece apart the date in `yyyymm` format in order to create a proper date using the `Date` object.

One annoying aspect of this is checking for null values and any function you create needs to be resilient to null values. In the code example below, we check for null values in the arguments using code like this:

```
// Check for nulls.
if (iMonthid == null || iShiftValue == null) {
 return -1;
}
```

If a null value is detected, the code immediate returns a `-1` to let the user know there was a problem. Note that there are several places where a `-1` is returned if something is not quite right, such as if the month value from the `yyyymm` argument is not within the range `1` to `12`.

Also, be aware that the `Date` object expects the year to be off of `1900`; that is, the year `1900` should be passed in as `0`. In this case, we subtract `1900` from the year passed into the function. The `Date` object expects the month value to start at `0` for January, `1` for February, … , `11` for December. In this case, we subtract `1` from the month.

In order to apply the shift value, we're using the `Calendar` object since it contains a spiffy `add()` method which works a treat with our shift value. Then, we use the `SimpleDateFormat` to create our final integer in `yyyymm` format. Yeah…uh…there's probably a better way to do this, but it works!

Finally, the code within the `evaluate()` method is ensconced in a `try-catch` block.

Here's the complete code listing:

```
import org.apache.hadoop.hive.ql.exec.UDF;
import java.util.*;
import java.text.SimpleDateFormat;
import org.apache.commons.lang3.StringUtils;

public class MonthIdShifter extends UDF {

 public MonthIdShifter() {
 }

 public int evaluate(Integer iMonthid,Integer iShiftValue) {

  try{

   // Check for nulls.
   if (iMonthid == null || iShiftValue == null) {
    return -1;
   }

   // Convert iMonthId and iShiftValue to String
   String sMonthId = iMonthid.toString();
   String sShiftValue = iShiftValue.toString();

   // Compute the length of the string...must be length of 6 (YYYYMM).
   if (sMonthId.length() != 6) {
    return -1;
   }

   // If iShiftValue is zero, just return the iMonthid
   if (iShiftValue == 0) {
```

```
        return iMonthid;
       }

       // Break up with YYYYMM into YYYY and MM. Set DD to 1.
       // Must subtract 1900 from year.  Must subtract 1 from month.
       int iYYYY = Integer.parseInt(sMonthId.substring(0,4)) - 1900; // YYYY - 1900
       int iMM = Integer.parseInt(sMonthId.substring(4)) - 1; // MM - 1
       int iDD = 1; // DD

       // iMM should be in the range of 0 to 11.
       if (iMM < 0 || iMM > 11) {
        return -1;
       }

       // Create a Date object.
       Date dMonthId = new Date(iYYYY,iMM,iDD);

       // Add iShiftValue to dMonthId.
       Calendar cal = Calendar.getInstance();
       cal.setTime(dMonthId);
       cal.add(Calendar.MONTH,iShiftValue);
       Date dMonthIdShifted = cal.getTime();

       // Get back YYYYMM from dMonthIdShifted.
       SimpleDateFormat DATE_FORMAT = new SimpleDateFormat("yyyyMM");
       int iYYYYMMShifted = Integer.parseInt(DATE_FORMAT.format(dMonthIdShifted));

       return iYYYYMMShifted;

      } catch(IllegalArgumentException exception) {

       return -1;

      } catch(Exception exception) {

       return -1;

      }

     }

    }
```

Now, despite the method being called `evaluate()`, the function that's actually called from ImpalaSQL is `MonthIdShifter()`, not `evaluate()`. It's the class name here that's important.

Naturally, you need to compile the program `MonthIdShifter.java`. At the command line, run the following to create the compiled Java class `MonthIdShifter.class`:

```
[smithbob@lnxserver ~]$ javac -cp $CLASSPATH:. MonthIdShifter.java
```

If any error messages are displayed during the compilation, make corrections and rerun the line above. If all goes well, the file `MonthIdShifter.class` will be created.

Next, you need to create a Java `.jar` file to contain the class `MonthIdShifter.class`. We'll call the `.jar` file `MonthIdShifter.jar`. Here's how to create the `.jar` file:

```
[smithbob@lnxserver ~]$ jar -cfv MonthIdShifter.jar MonthIdShifter.class
```

Next, you need to copy the `.jar` file over to your UDF directory in HDFS (run this code on one line, bruh):

```
[smithbob@lnxserver ~]$ hadoop fs -copyFromLocal
                                   -f
                                   /home/smithbob/UDFDevel/MonthIdShifter.jar
                                   /data/prod/teams/prod_schema/UDF/MonthIdShifter.jar
```

The `-f` option will overwrite `MonthIdShifter.jar` if it already exists in the HDFS destination location.

Now, in order to tell ImpalaSQL that your brand new function actually exists, you have to issue the `CREATE FUNCTION` statement in `impala-shell`. Make sure to change to the database where you want the function to be defined first!

```
CREATE FUNCTION IF NOT EXISTS MONTHIDSHIFTER(INT,INT)
 RETURNS INT
 LOCATION '/data/prod/teams/prod_schema/UDF/MonthIdShifter.jar'
 SYMBOL='MonthIdShifter';
```

The `SYMBOL` value is the name of the class within the Java `.jar` file you want associated with the ImpalaSQL function name on the `CREATE FUNCTION` line (`MONTHIDSHIFTER`, here).

Next, you can test your UDF using code similar to the following:

```
[hdpserver:21000] prod_schema> select monthidshifter(201712,12);

+-------------------------------------+
| monthidshifter(201712, 12)          |
+-------------------------------------+
| 201812                              |
+-------------------------------------+
```

Recall that you can get a list of your own UDFs from the `impala-shell` command line by running the `show functions` command:

```
[hdpserver:21000] prod_schema> show functions;

+--------------+--------------------------------+-------------+---------------+
| return type  | signature                      | binary type | is persistent |
+--------------+--------------------------------+-------------+---------------+
| INT          | monthidshifter(INT, INT)       | JAVA        | true          |
+--------------+--------------------------------+-------------+---------------+
```

Note that the `CREATE FUNCTION` Statement shown above is not permanent and if, say, the cluster is rebooted, your function will disappear…poof! To ensure that your function is permanent, unlike love and scotchguarding, use the following syntax instead:

```
CREATE FUNCTION IF NOT EXISTS MONTHIDSHIFTER
 LOCATION '/data/prod/teams/prod_schema/UDF/MonthIdShifter.jar '
 SYMBOL='MonthIdShifter';
```

And, as usual, if you're having problems, please contact your extra-galactic Hadoop Administrator.

PART IX - Appendages

Appendage #1 – Hadoop Administrator E-Mail

Below is the full Hadoop Administrator E-Mail. Don't forget that you can find it as well as all of the code in the book on my personal (yet stupidly named) website `www.sheepsqueezers.com`.

Hadoop Administrators:

Tally Ho! My name is Bob Smith and I work for the *<insert dept name here>* department and, as you may have heard, I've been tasked with moving data off our legacy *<insert legacy database name>* database to the Hadoop database. I was hoping that you could be my contact for the duration of this conversion.

First, thank you up-front for helping out since this Hadoop shizz is new to me and my team.

Second, you probably won't be surprised that I have about a bazillion questions for you which I've placed below. Your responses will go a long way in helping me and my team move to Hadoop as quickly (and painlessly!) as possible.

Here goes...

☐ Do you have a Linux edge node server that my team can use? If so, what's the server's host name? My team and I will be automating some processes using Linux scripts, so access to a Linux edge node server will help us out greatly.

☐ My team and I plan to use PuTTY to connect to the Linux edge node server. I just want to confirm that we must use port 22 (SSH) when setting up a connection to the edge node server. Do you recommend something other than PuTTY?

☐ On our legacy database, the schema we use is named *<insert name of legacy database schema name here>*. Can you please set up the same schema name on the Hadoop database?

☐ Since my team and I will use the edge node server as well as the Hadoop database, can you please set up the following individuals with an account on the Linux edge node server as well as access to the Hadoop database schema requested above? *<insert your Team's corporate e-mail addresses here>*

Also, the following team members should be given privileged access to run Hadoop commands via hadoop/hdfs from the Linux command line: *<insert select team members who should have higher privileges, including yourself, here>*

☐ Not all of my team members are highly technical, but would like to run simple queries against the Hadoop database. Do you have the Hadoop database web interface **Hue** set up and accessible? If so, what's the URL?

☐ In order to kill runaway SQL queries, can you please list the URLs to the Hadoop query webpages? I believe these URLs generally use port 25000 (/queries), but don't hold me to that...I'm new to these parts.

☐ Can you recommend a SQL client application (such as Toad Data Point, DBeaver, SQuirreL, etc.) for use with Hadoop? What do you use?

- ☐ Do you have Hive and Impala ODBC (32-bit/64-bit) and JDBC drivers available on the corporate network? If so, I'd like to access them so that I may set up my team's SQL client software (among other things). If not, can you recommend where I may download these drivers?

- ☐ Speaking of ODBC and JDBC drivers, can you please provide example connection information/strings for both ODBC and JDBC connections to Hive (port 10000?) as well as Impala (port 21050?)? We'll be using the ODBC connection information with applications such as Microsoft Excel, PowerBI, Tableau, etc. The JDBC connection strings will be used with client software that uses JDBC rather than ODBC such as DBeaver, SQuirreL, etc.

- ☐ Does our corporate network run Kerberos? If so, when creating cron jobs to run automatically, we may need to create a keytab file containing Kerberos-related information. Which encryption types do you suggest we include in the keytab file? arcfour-hmac-md5? aes256-cts? rc4-hmac? Any others? Also, what's our **Kerberos Realm** and **Host FQDN**? If not Kerberos, then LDAP?

- ☐ We would like the ability to access our legacy database (*<insert name of legacy database>*) from the Linux edge node server for use with sqoop and other tools. Can you please install the software necessary so that my team and I may access the legacy database from there?

- ☐ Is there a generic account available on the Linux edge node server for me and a few of my team members to use? We'd like a single account to execute our production code. If so, can you please forward the username and password? If not, can you please create an account on the Linux edge node server whose password is static? Also, please give this account access to the appropriate schemas as well as hadoop/hdfs privileges.

- ☐ Is HPL/SQL available from the Linux edge node server? If not, can you please install it so that my team and I can create and execute procedures on the Linux edge node server against the Hadoop database? Also, where is the file hplsql-site.xml located?

- ☐ Is there a directory on the Linux edge node server where we can store the team's production code? If not, can you please create a directory accessible by my team as well as the generic account?

- ☐ Can you please create a directory in HDFS specifically for me and my team for use with external tables? Something like **hdfs://hdpserver/data/prod/teams/*<schema>*** or whatever your standard is.

- ☐ I feel completely comfortable downloading and maintaining many of my department's dimension tables, but some of the fact tables are quite large. I'm hoping you can intercept the process involved in importing the fact tables and incorporate them into your process. Can we have a conversation about that?

- ☐ What are the version numbers for the following?

  - Linux (on the edge node server)
  - Apache Hadoop
  - Hive
  - Impala
  - HPL/SQL
  - Hive ODBC Driver
  - Impala ODBC Driver

- Hive JDBC Driver
- Impala JDBC Driver

☐ Can you please install the Linux utility dos2unix on the Linux edge node server? Since our laptops are Windows-based, we may need to convert files using dos2unix.

☐ Which Thrift Transport Mode should we be using? SASL? Binary? HTTP?

☐ Does the Hadoop Database use Secure Sockets Layer (SSL) for connections? When I go to set up an ODBC connection, there's an option asking whether I should enable SSL. Should I?

☐ My team and I will be using the storage formats TEXTFILE, PARQUET and KUDU almost exclusively. Can you please indicate the SQL CREATE TABLE options required to use the KUDU storage format, if any? Can you recommend the number of partitions we should use with KUDU tables? Do we have to include the table property **kudu.master_addresses** in our SQL code? If so, can you include an example of this?

☐ In our legacy *<insert name of legacy database>* database, we have access to useful metadata such as table names, column names, data types, etc. within the database via ALL_TABLES, ALL_TAB_COLUMNS, INFORMATION_SCHEMA, etc. Can you create a view or views to mimic this from within the Hadoop database accessible from our new database schema? If not, can you give us read-only access to the underlying MetaStore database's metadata tables/views?

☐ Does the version of ImpalaSQL installed on the Hadoop database include the extensions to GROUP BY such as CUBE, ROLLUP, GROUPING SETS, etc.?

☐ Is Apache Spark installed on the Linux edge node server? If so, what's the version number? As I would like to use Spark with Python, is pyspark available to use?

☐ My Team and I may create one or more user-defined functions (UDFs) for Impala. Can you create a directory in HDFS where we may place our Java .jar files? Also, can you update the PATH and CLASSPATH so that we have access to java and javac?

Thanks,
Bob Smith

Appendage #2 – Linux on Windows

As we alluded to earlier in the book, you can repurpose an old laptop by installing Linux on it, install virtual machine software (such as VMWare or VirtualBox) to have Linux features available from your Windows laptop, or install Linux utilities directly on your Windows laptop. There are several roads less traveled to go down:

The **first method**, repurposing an old laptop, means that the entire machine would be taken over by the Linux operating system. Yes, you can set up your laptop to *dual boot* with one side being Linux and the dark side being Windows, but that's a fairly complex process, so we won't discuss that here. You can either find instructions on how to do that on the web, or just invite Bill Gates and Linus Torvalds over for dinner.

The **second method**, installing virtual machine software, is a great choice if you have enough RAM and disk space available to run both the Linux virtual machine as well as everything else on your Windows laptop. This is a fairly simple process and you're not obliged to run your freshly minted Linux virtual machine all of the time: you can start and stop it at will ("*FIRE AT WILL!*" "*But I don't know any of their names!*"). We won't discuss that here.

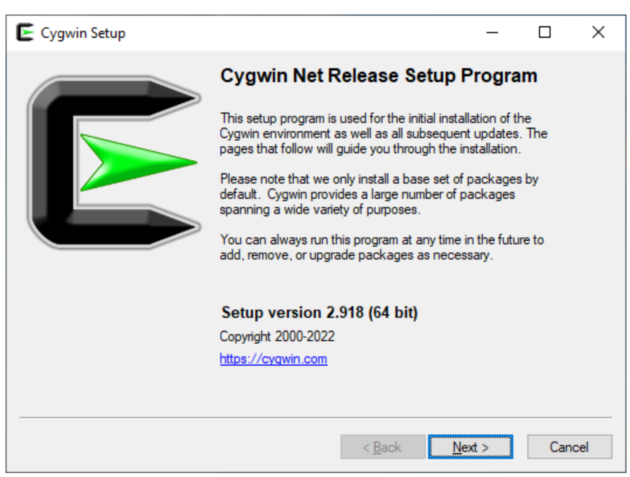
The **third method** is to install a Linux port known as Cygwin on your Windows laptop. Effectively, familiar Linux utilities such as `ls`, `cp` and `rm`, etc. are available from the Windows Command Prompt along with their Windowy counterparts `dir`, `copy` and `del`, etc. without the need for a virtual machine. We discuss this in the next section.

The **fourth method** is to install Windows Subsystem for Linux (WSL), provided by Microsoft, on your Windows laptop. This is similar to Cygwin, but much more powerful. We discussed this briefly in *Chapter 5 – Creating Your Very Own Hadoop Playground*, and this was recently discussed in the article "Linux Loves Windows" in the January 2022 edition of MaximumPC magazine (yes, I do have a subscription…thpppttt!), so have at it!

Installing Cygwin

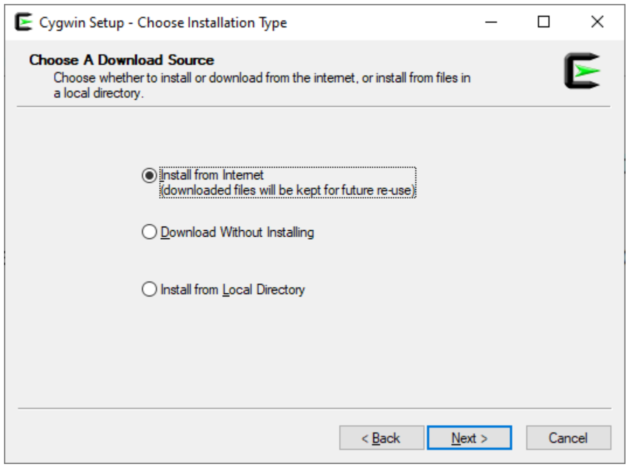
Installing Cygwin is a fairly straight-forward process:

- ☐ Create a folder named `cygwin` in `\\corp\dept\software` (or other directory) to contain the downloaded installation software.
- ☐ Navigate your browser to **`https://www.cygwin.com/`**. Under the section labeled **Installing Cygwin**, right-click on `setup-x86-64.exe` and save it to the `cygwin` folder.
- ☐ Scan the downloaded file with your anti-herpes software.
- ☐ Right-click on the file, click on Properties menu item from the popup menu. On the Properties dialog box, ensure the checkbox to the left of **Unblock** is checked and then click Apply. Click OK to dismiss the dialog.
- ☐ Right-click on the file and click on **Run as administrator** menu item from the popup menu. This will start the setup program.
- ☐ Click the Yes button if the User Account Control dialog box appears.
- ☐ The **Cygwin Setup** dialog box will appear:



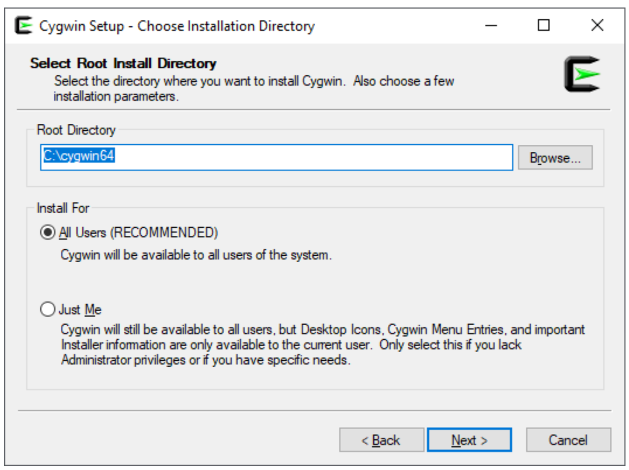
- ☐ Click Next.

☐ On the **Choose Installation Type** dialog box, ensure the radio button to the left of the text **Install from Internet** is checked.

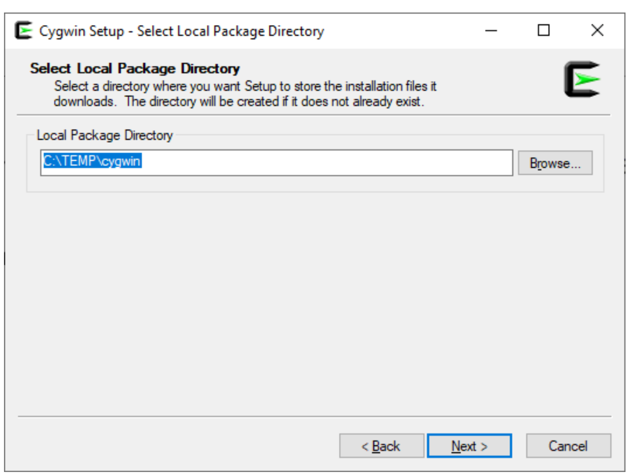


☐ Click Next.

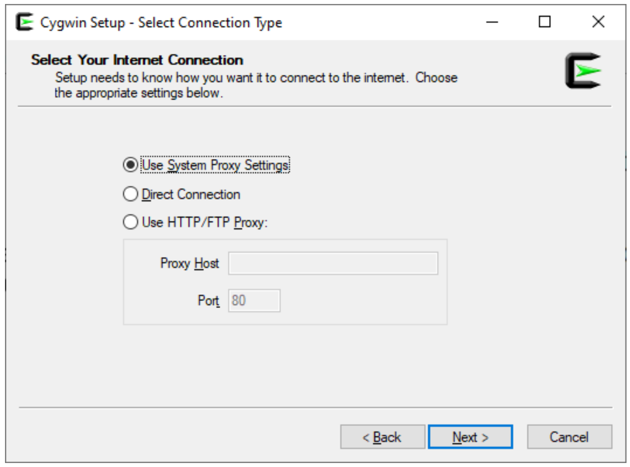
☐ On the **Choose Installation Directory** dialog box, accept the defaults and click Next.



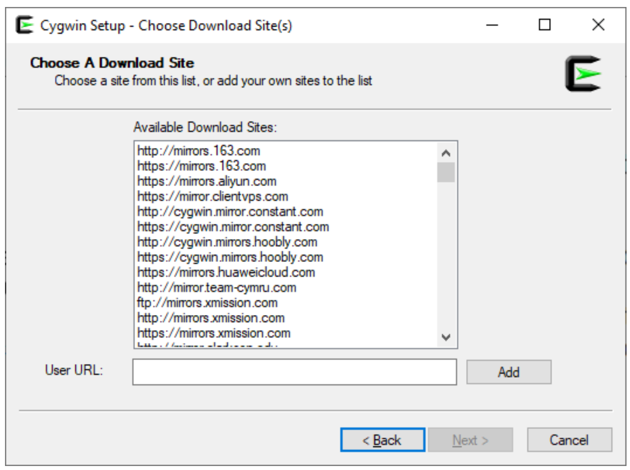
☐ On the **Select Local Package Directory** dialog box, accept the default and click Next.



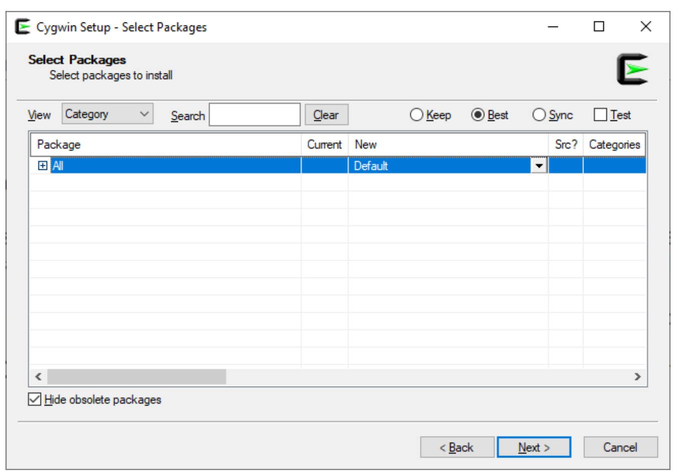
☐ On the **Select Connection Type** dialog box, ensure that the radio button to the left of the text **Use System Proxy Settings** is checked. Note that if this fails, you may need to switch to **Direct Connection** instead. Click Next.



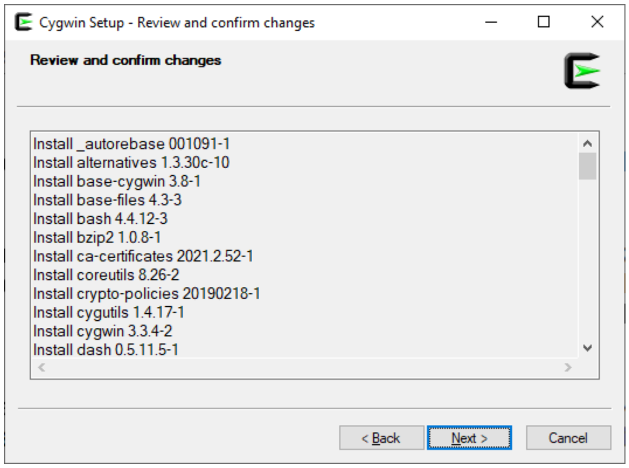
☐ On the **Choose a Download Site** dialog box, select a download site and click Next.



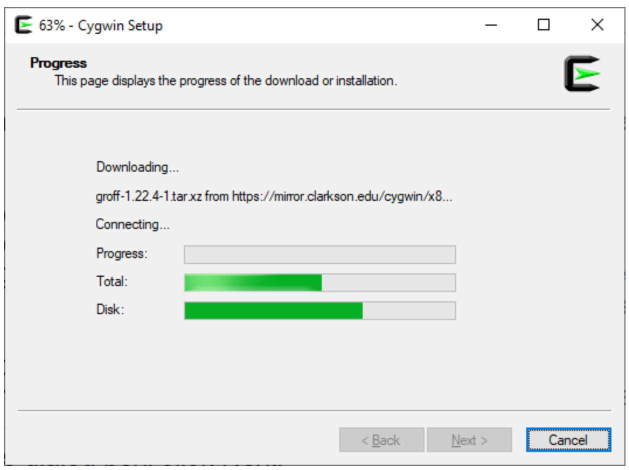
☐ On the **Select Packages** dialog box, ensure the radio button to the left of the text Best is checked and click Next. Note that this will install quite a bit of software, but YOLO!



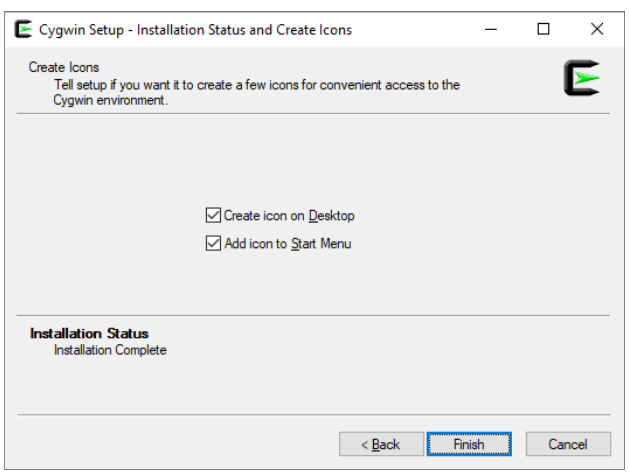
☐ On the **Review and confirm changes** dialog box, click Next.



- ☐ At this point, the **Cygwin Setup** dialog will display fascinating installation statistics…*yawn!*...so best to go grab a sandwich and come back later when the dust has settled.



- ☐ On the **Installation Status and Create Icons** dialog box, accept the defaults and click Finish.



With the installation process complete, you can double-click the **Cygwin64 Terminal** icon on your desktop to start the Cygwin terminal. Once started, you'll see the following output in the terminal window:

```
Copying skeleton files.
These files are for the users to personalise their cygwin experience.

They will never be overwritten nor automatically updated.

'./.bashrc' -> '/home/smithbob//.bashrc'
'./.bash_profile' -> '/home/smithbob//.bash_profile'
'./.inputrc' -> '/home/smithbob//.inputrc'
'./.profile' -> '/home/smithbob//.profile'

smithbob@DESKTOP-A1BCD2E ~
$
```

The dollar sign indicates the Linux command prompt is ready to accept Linux commands just like those outlined in Part II, *Introduction to the Linux Operating System*.

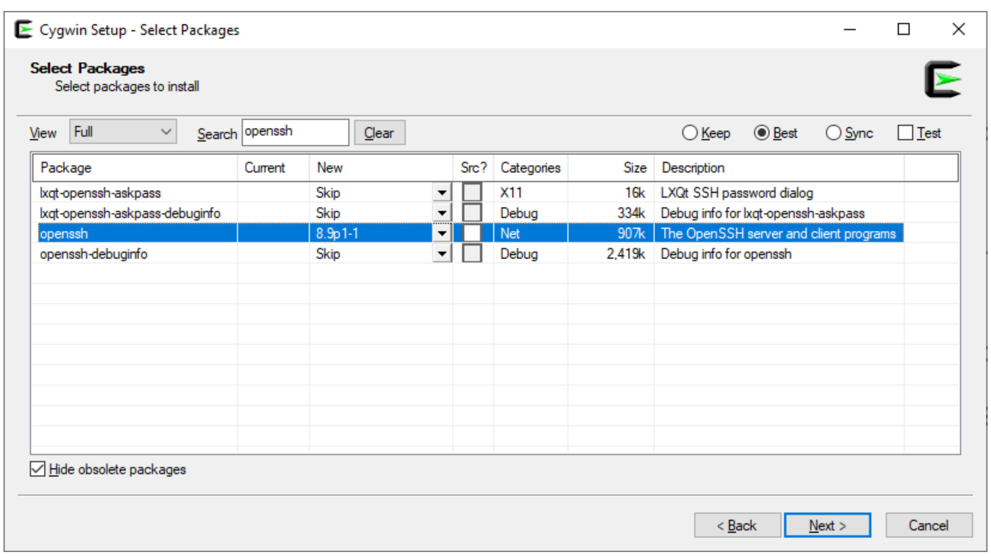
Note that Cygwin gives you access to the Windows file system, such as your C-Drive. To do this, change directory to `/cygdrive/c`: **`cd /cygdrive/c`**. At this point, the working directory is now at the C-Drive.

Installing Additional Features

Not every feature is installed by default and you'll have to install them after the ball has ended, Cinderella. You can use the method outlined in this section to install additional features.

For example, in *Chapter 36 – Using ssh and scp from Linux and Windows*, we discuss using `ssh` from the Windows Command Prompt, but this may not be available on your Windows installation. If that's the case, you can install it in Cygwin. To do this, perform the following steps:

- ☐ Start the Cygwin setup program exactly the same way as outlined in the previous section
- ☐ Click the Next button until you reach the **Select Packages** dialog box
- ☐ Ensure the drop-down box to the right of the text **View** is set to Full
- ☐ In the input box to the right of the text **Search**, type in `openssh` and the list of related packages should display automatically
- ☐ To the right of the package `openssh`, change the drop-down box from **Skip** to the latest version (**`8.9p1-1`** is shown in the image below)



- ☐ Click Next
- ☐ On the **Review and confirm changes** dialog box, click Next
- ☐ At this point, the additional software will be installed.

- ☐ On the **Installation Status and Create Icons** dialog box, uncheck both boxes and click Finish.

At this point, the secure shell utility `ssh` is available from the Cygwin Terminal. See *Chapter 36 – Using ssh and scp from Linux and Windows* for more on `ssh` and `scp`.

Appendage #3 – When HPL/SQL Causes You Pain

In PART V, *HPL/SQL Procedural Language*, we mentioned that, occasionally, HPL/SQL won't work straight *outta da box* and we present some possible solutions in this appendage. The following are the two main ways HPL/SQL can be made available on your Linux edge node server:

- ☐ HPL/SQL Pre-Installed – A version of HPL/SQL comes pre-installed with your flavor of Hadoop (Cloudera or other company).
- ☐ HPL/SQL Downloaded – You downloaded HPL/SQL from the website `http://www.hplsql.org/` and unpacked it into a local directory.

In both cases, HPL/SQL either doesn't work at all, or works with Impala connection issues. Recall that the file `hplsql-site.xml` contains connections to Hive, Impala, MySQL, etc. which you or your Hadoop Administrator have updated with the appropriate JDBC connection information. No matter the problems you're having, this file still needs to be updated before you can connect to, at the very least, Impala.

In the first case above, the `hplsql-site.xml` file is not being found. In the second case, the Java `CLASSPATH` needs to be added/updated in the `hplsql` script in order for important Java `.jar` files to be found. We discuss both below.

In all cases, you'll need the Impala JDBC drivers to be downloaded and placed into an appropriate directory on the Linux edge node server. Without these drivers, you cannot connect to Impala via HPL/SQL, never ever ever. We discuss this below as well.

HPL/SQL Pre-Installed

If HPL/SQL is pre-installed on your flavor of Hadoop, you may need to perform the steps outlined below if the `hplsql` executable functions properly, but your own `hplsql-site.xml` file is not being found causing an inability to connect to Impala, Hive, etc. Recall that the `hplsql-site.xml` file should be picked up in the location where `hplsql` is executed. This problem may stem from the fact that a version of the `hplsql-site.xml` file is actually embedded in one of the associated Java `.jar` files causing your own `hplsql-site.xml` file to be completely ignored, like you at that high school dance. In order to correct this, you must locate the offending `.jar` file, unpack it, delete the `hplsql-site.xml` file, recreate the Java `.jar` file excluding the `hplsql-site.xml` file, and then, finally, copy over the new `.jar` file to the appropriate location. Oh boy oh boy oh boy!!

At least for the Cloudera version of Hadoop installed on my test machine, the `hplsql` script is located in the directory `/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/lib/hive/bin`. This script makes use of the Java `.jar` file `hive-hplsql.jar` located in the directory `/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/lib/hive/lib`. This actually is a link pointing to the Java `.jar` file `hive-hplsql-3.1.3000.7.1.7.0-551.jar` which is actually located in the directory `/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/jars`. I know, I know…that's a lotta redirection!

So, copy the file `hive-hplsql-3.1.3000.7.1.7.0-551.jar` (or your particular version) to an empty directory:

```
cd /home/smithbob
mkdir tmp
cd tmp
cp /opt/cloudera/ parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/jars/hive-hplsql-
3.1.3000.7.1.7.0-551.jar  hive-hplsql-3.1.3000.7.1.7.0-551.jar
```

Next, since this file is a Java archive file, or jar file, you can use the Java `jar` utility to unpack its contents into the `tmp` directory:

```
jar -xvf hive-hplsql-3.1.3000.7.1.7.0-551.jar
```

Change to the hive-hplsql-3.1.3000.7.1.7.0-551 directory. You'll note that the file hplsql-site.xml is in this directory along with other files and folders. Delete the file hplsql-site.xml with all the rage of a thousand camels now!! Also, you can delete the file ../hive-hplsql-3.1.3000.7.1.7.0-551.jar file as well since you'll recreate it below.

Now that both hplsql-site.xml and hive-hplsql-3.1.3000.7.1.7.0-551.jar have been disposed of James Bond stylie, we can recreate hive-hplsql-3.1.3000.7.1.7.0-551.jar. At the Linux command line, issue the following command (ensure you're in the hive-hplsql-3.1.3000.7.1.7.0-551 directory):

```
jar -cvf ../hive-hplsql-3.1.3000.7.1.7.0-551.jar ./*
```

Once complete, the recreated Java .jar file hive-hplsql-3.1.3000.7.1.7.0-551.jar should be located in /home/smithbob (take note of the two dots in the command above).

Now, working with your Hadoop Administrator, the following tasks should be completed:

1. Rename /opt/cloudera/ parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/jars/hive-hplsql-3.1.3000.7.1.7.0-551.jar to /opt/cloudera/ parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/jars/hive-hplsql-3.1.3000.7.1.7.0-551.jar**_ORIG**
2. Copy hive-hplsql-3.1.3000.7.1.7.0-551.jar from /home/smithbob/ to /opt/cloudera/ parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/jars

At this point, when you run hplsql, your local hplsql-site.xml should be picked up. Note that the Impala JDBC Drivers still need to be installed prior to connecting to the database via Impala. See below for more.

HPL/SQL Downloaded

If you or your Hadoop Administrator downloaded HPL/SQL from the site http://www.hplsql.org/, you'll still need to make modifications to the associated hplsql script. The latest version available on the website is 0.3.31. As stated earlier in the book, your Hadoop Administrator can locate the HPL/SQL directory in one central location, or it can be placed in /etc/skel so that each user can have his/her own copy when each account is created. The hplsql script contains the following code:

```
#!/bin/bash

export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop/*"
export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop/lib/*"
export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/etc/hadoop/conf"

export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop-mapreduce/*"
export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop-mapreduce/lib/*"

export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop-hdfs/*"
export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop-hdfs/lib/*"

export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop-yarn/*"
export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop-yarn/lib/*"

export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hive/lib/*"
export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hive/conf"

export HADOOP_OPTS="$HADOOP_OPTS -Djava.library.path=/usr/lib/hadoop/lib/native"

SCRIPTPATH=${0%/*}

java -cp $SCRIPTPATH:$HADOOP_CLASSPATH:$SCRIPTPATH/hplsql-0.3.31.jar:$SCRIPTPATH/antlr-runtime-
4.5.jar $HADOOP_OPTS org.apache.hive.hplsql.Hplsql "$@"
```

Take note of the emboldened code after the Java classpath switch cp. You'll have to place additional Java .jar files after antlr-runtime-4.5.jar. The additional Java .jar files are as follows:

- ☐ hive-jdbc-3.1.3000.7.1.7.0-551-standalone.jar

- ☐ `hive-exec.jar`
- ☐ `hive-jdbc.jar`
- ☐ `libthrift-0.9.3-1.jar`
- ☐ `httpcore-4.4.10.jar`
- ☐ `httpclient-4.5.6.jar`
- ☐ `hadoop-common.jar`
- ☐ `hadoop-hdfs.jar`
- ☐ `hadoop-auth.jar`
- ☐ `commons-cli-1.4.jar`
- ☐ `commons-io-2.4.jar`
- ☐ `hadoop-core-2.6.0-mr1-cdh5.16.2.jar`
- ☐ `commons-logging-1.1.1.jar`
- ☐ `hadoop-hdfs-client-3.1.1.7.1.7.0-551.jar`
- ☐ `commons-collections-3.2.2.jar`

Note that these Java `.jar` files may appear in several different locations on the Linux edge node server. Also, be aware that using the most recent version of these `.jar` files may seem like the way to go, but it's not always the best policy. Please work with your super-brainy Hadoop Administrator to find the appropriate locations as well as versions.

Note, also, that you'll need to include `ImpalaJDBC4.jar` as well in order to connect to Impala. We talk about this in the next section. A similar comment goes for other connections you'd like to make such as to MySQL, Oracle, and so on. Those associated Java JDBC `.jar` files need to be placed in a central location so that they can be referenced in the code below.

Now, the `hplsql` script, at least **for me** with my locations/versions, looks like this:

```
#!/bin/bash

export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop/*"
export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop/lib/*"
export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/etc/hadoop/conf"

export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop-mapreduce/*"
export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop-mapreduce/lib/*"

export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop-hdfs/*"
export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop-hdfs/lib/*"

export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop-yarn/*"
export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hadoop-yarn/lib/*"

export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hive/lib/*"
export "HADOOP_CLASSPATH=$HADOOP_CLASSPATH:/usr/lib/hive/conf"

export HADOOP_OPTS="$HADOOP_OPTS -Djava.library.path=/usr/lib/hadoop/lib/native"

SCRIPTPATH=${0%/*}

java -cp .:$SCRIPTPATH:$HADOOP_CLASSPATH:$SCRIPTPATH/hplsql-0.3.31.jar:$SCRIPTPATH/antlr-runtime-
4.5.jar:/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/jars/hive-jdbc-3.1.3000.7.1.7.0-551-
standalone.jar:/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/lib/hive/lib/hive-
exec.jar:/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/lib/hive/lib/hive-
jdbc.jar:/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/lib/hive/lib/libthrift-0.9.3-
1.jar:/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/lib/hive/lib/httpcore-
4.4.10.jar:/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/lib/hive/lib/httpclient-
4.5.6.jar:/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/lib/hadoop/hadoop-
common.jar:/opt/cloudera/cm/lib/cdh5/mr1/hadoop-core-2.6.0-mr1-
cdh5.16.2.jar:/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/lib/hadoop-hdfs/hadoop-
hdfs.jar:/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/lib/hadoop/client/hadoop-
auth.jar:/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/jars/commons-cli-
1.4.jar:/opt/cloudera/parcels/CDH/jars/commons-io-2.4.jar:/opt/cloudera/cm/lib/commons-logging-
1.1.1.jar:/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/jars/hadoop-hdfs-client-
3.1.1.7.1.7.0-551.jar:/opt/cloudera/parcels/CDH-7.1.7-1.cdh7.1.7.p0.15945976/jars/commons-
collections-3.2.2.jar:/home/smithbob/jars/ImpalaJDBC4.jar $HADOOP_OPTS
org.apache.hive.hplsql.Hplsql "$@"
```

Don't forget to place your hplsql-site.xml file in the same location as the hplsql script. Now, when you run an HPL/SQL program, you'll see the following messages :

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further
details.
```

The reason we updated the classpath in the hplsql script was to avoid any error messages caused by both slf4j and log4j being found in the CLASSPATH. The messages above are just informational and can be ignored. HPL/SQL should work fine now.

Impala (and Other) JDBC Drivers

As indicated several times above, you'll have to download the JDBC drivers for Impala, MySQL, etc. in order to connect to these databases. You or your Hadoop Administrator can download these from the InterWebs. For example, you can download the Impala JDBC driver files from the Cloudera website https://www.cloudera.com/downloads/connectors/impala/jdbc. There are several version available in the downloaded zip file, such as ImpalaJDBC4.jar, ImpalaJDBC41.jar and ImpalaJDBC42.jar. Don't forget to place these .jar files in a central location where Hadoop and HPL/SQL can find them. You'll need to add these Java .jar files to the classpath cp switch in the prior section. At this point, you should be able to successfully connect to Impala, MySQL, etc. within an HPL/SQL program.

Appendage #4 – When Bad Errors Happen to Good Programmers

In this appendage, we discuss some error messages you may receive during the course of working with Hadoop, ImpalaSQL, the storage formats, etc. and potential solutions to them. Note that the error messages here aren't presented in any particular order because that would be logical and make sense.

`CREATE TABLE` Statement Fails After `DROP TABLE` Statement

Occasionally, you may receive the following error message after you issue a `CREATE TABLE` Statement following a supposedly successful `DROP TABLE` Statement:

`ERROR: AnalysisException: Table already exists:` ***`database_name.table_name`***

This may be caused by several obvious reasons, such as a misspelling, which can be easily rectified. Barring these simple mistakes, recall that your Hadoop cluster is composed of several servers (nodes) and each one has to register that the `DROP TABLE` Statement succeeded. If something goes wrong, Hadoop won't believe the table has been completely smashed to smithereens and the subsequent `CREATE TABLE` Statement will fail with the error message above. I guess one way…what's that noise?...oh! I'm getting a text from my Mother…one second…

It's your Mother. I'm still alive.

I know, Ma. If you were dead, Dad would have me take him to a bar to scope for chicks.

Your Father? Scope for chicks? The man smells of formaldehyde.

So, what's up, Ma? You don't usually text me.

I know, but I just heard on the news the entire logging industry is causing you computer programmers problems. Do you need my help, dear?

Huh? The logging industry? Oh! No, Ma, it's software called log4j that's the problem. We're all fine with the logging industry here.

Oh, that is nice to hear. I don't want your legs broken. So, how's work otherwise?

Fine, I guess. I'm having trouble re-creating a table after I dropped it. It's baffling. I don't expect you to understand.

Have you tried setting the option SYNC_DDL to 1 in your code prior to running any SQL statements?

WHAT? WHAT? WHAT? How do you know that?

I've been married to your father for 35 years, so I read random stuff on the InterWebs to keep from speaking to him.

Thanks, Ma! I'll give that a shot.

That's lovely, dear. Would you like to come over for dinner tonight?

I'd love to, Ma! Uh, Ma, what's with the emojis?

Oh, I'm making eggplant parmesan with a side salad.

Oy.

One way to fix this issue, as the elderly Mrs. Smith mentioned above, is to tell Hadoop you want to wait until the entire cluster reports in with a laurel-and-hardy handshake before moving on to the next SQL statement. This may cause a slight delay, but that's better than having your program bomb or the logging industry break your legs (you can't run…you can't hide…the logging industry has long branches!). Here's an example:

```
SET SYNC_DDL=1;
USE PROD_SCHEMA;

DROP TABLE IF EXISTS DIM_US_POSTAL_CODE PURGE;
CREATE TABLE DIM_US_POSTAL_CODE(...snip...);
```

But, please have a conversation with your lovely Hadoop Administrator about this issue before randomly setting this option. If not, you'll have the entire Hadoop Administrator industry to deal with!! Oy vey!!

System Metadata Delay to Table Creation

Although we talked about how Hadoop handles database metadata earlier in the book, one **very unlikely** problem you may have is that the Hadoop database has a delayed reaction to the creation of a table. That is, you create a table, but the metadata is not immediately available, such as when you describe the table. Recall that Hadoop metadata is not actually stored in the Hadoop database itself, but outside of the Hadoop database in a completely different database such as MySQL, PostgreSQL, etc. Normally, this wouldn't be a problem if the metadata database is located near in proximity to your Hadoop database. But, if the metadata is located in a database in another part of the country or the world, then there's the potential for a sizeable delay (well…many milliseconds, anyway) to occur. If you suspect this is happening, please have a conversation with your brilliant Hadoop Administrator. Note that this problem isn't limited to just the creation of a table, as you can well imagine.

Incompatible Parquet Schema Warning

You may occasionally receive the following warning message when querying a Parquet table from ImpalaSQL:

```
WARNING: File
'hdfs://hdpserver/data/prod/teams/prod_schema/table_name/file_name.parq
' has an incompatible Parquet schema for column
'prod_schema.table_name. column_name'. Column type: XXX, Parquet
schema: optional XXX column_name [i:## d:## r:##]
```

This occurs because Impala doesn't currently support all of the data types (as well as data ranges) that the Parquet format itself supports. The solution is to modify the problematic column's data type to match the data type Impala recognizes. Be aware that Parquet files can be created outside of the Hadoop environment with other software and attempting to use these files may also cause this warning to appear.

Invalid Parquet Version Number Error

You may occasionally receive the following error message when working with a Parquet table from ImpalaSQL:

```
ERROR: File 'hdfs://hdpserver/directory/.../table-name' has an invalid
Parquet version number:   #
```

This can occur for a variety of reasons, but one solution is to perform a refresh on the offending table via ImpalaSQL: `REFRESH schema.table-name;`. After performing the refresh, try accessing the table once more. If you still receive this error, please contact your Hadoop Administrator.

Note that if the offending table is located in a schema other than your own, please contact your Hadoop Administrator before issuing the refresh to alert him/her of the issue.

[Also, be aware that there may be one or more null characters before the pound sign in the error message above. So, if your editor fails to copy the error message text complaining it can't copy null characters, just delete what appears to be spaces after the colon (`:`) and before the pound sign (`#`) and try to copy the message again.]

Block Locality Metadata Warning

You may occasionally receive the following error message when accessing one or more tables from ImpalaSQL:

```
WARNING: Read # MB of data across network that was expected to be
local. Block locality metadata for table 'schema.table' may be stale.
This only affects query performance and not result correctness. One of
the common causes for this warning is the HDFS rebalancer moving some
of the file's blocks. If this issue persists, consider running
"INVALIDATE METADATA `schema`.`table`".
```

This can occur for a variety of reasons, but one solution is to perform the recommended `INVALIDATE METADATA` on the offending table via ImpalaSQL: `INVALIDATE METADATA schema.table-name;`. After performing the invalidate, try accessing the table once more. If you still receive this error, please contact your Hadoop Administrator.

Note that if the offending table is located in a schema other than your own, please contact your Hadoop Administrator before issuing the invalidate to alert him/her of the issue.

Invalid Numeric Literal with Partitions

Recall we talked about partitions in *Chapter 16 – SQL Performance Improvements*. Occasionally, you may receive the following error message:

```
AnalysisException: Failed to load metadata for table: 'table-name'
CAUSED BY: TableLoadingException: Failed to load metadata for table:
table-name CAUSED BY: CatalogException: Invalid partition key value of
type: INT CAUSED BY: AnalysisException: invalid numeric literal:
literal-value-displayed-here CAUSED BY: NumberFormatException: null
```

This error is caused by attempting to insert data into the the partitioned table named `table-name`, but the partition's data type and the partition's value don't agree. The solution is to ensure that the data type specified for both the partition key and the data being inserted into the table match.

Authorization Error (User Does Not Have Privileges)

When first starting to use Hadoop, you may receive the following error message more than once:

```
ERROR: AuthorizationException: User 'smithbob@COMPANY.COM' does not
have privileges to...
```

Please work with your Hadoop Administrator to obtain the correct privileges for the underlying object(s).

Unknown Host Specified When Using `sqoop`

If you receive a variation on the following error message when using `sqoop` to pull from a remote database...

```
Error: java.lang.RuntimeException: java.lang.RuntimeException:
java.sql.SQLException: Unknown host specified
```

...there may be an issue communicating with the remote database from the Linux edge node server. Please contact your Hadoop Administrator who may want to bring in both the Network Administrator and the remote Database Administrator to determine the cause of the problem. Attempt to ping the remote server and paste the results into an e-mail as it may help the administrators. (If you're using Oracle, attempt to `tnsping` the remote database and place those results into the e-mail as well.)

Unreachable `impalad(s)`

If you receive the following error message while running an ImalaSQL query...

```
impala.error.OperationalError: Cancelled due to unreachable impalad(s):
impala-hostname.COMPANY.COM:22000
```

...your Hadoop Administrator may want to check if there's a space issue on the cluster. Once the space issue has been resolved, resubmit you SQL query. If the error still persists, your Hadoop Administrator may need to reboot Impala.

Server Unexpectedly Closed Network Connection (PuTTY)

If you receive the popup dialog box with the title **`PuTTY Fatal Error`** and message **`Server unexpectedly closed network connection`**, you most likely don't have permission to log into that server or the connection to the server has been severed. Please check the IP address/host name to ensure it's correct. If so, contact your Hadoop/Linux Administrator to get access to that server.

Multiple `COUNT(DISTINCT)` Statements Fail

Depending on the version of Impala you're running, you may receive the following error message when you issue more than one `COUNT(DISTINCT column_name)` in a SQL query:

```
Error while executing a query in Impala: [HY000] : AnalysisException:
all DISTINCT aggregate functions need to have the same set of
parameters as count(DISTINCT column_name); deviating function:
count(DISTINCT column_name)
```

Older versions of Impala prevent you from issuing more than one `COUNT(DISTINCT column_name)` in a SQL query, whereas newer versions allow for it. If you're site is running an older version of Impala, you can reformulate your SQL query to use several subqueries (or `WITH` Clauses) containing separate `COUNT(DISTINCT column_name)`'s and then join them together.

multiple count distincts
in a single query
fails a'ight, it just tanks
sigh, you become weary.
but! run a new version

yada yada yoo-yoo
works like a spell potion
so skip this verse, do-do

Error Creating Kudu Table

When creating a Kudu table, you may receive the following error message...

```
ERROR: ImpalaRuntimeException: Error creating Kudu table
'impala::prod_schema.table-name'
CAUSED BY: NonRecoverableException: Too many attempts
```

If your `CREATE TABLE` code for the `KUDU` table specifies the `kudu.master_address` in `TBLPROPERTIES`, remove it and try again. If the error still persists, please contact your Hadoop Administrator.

Memory is, Like, Oversubscribed, Dude!

When running a query in ImpalaSQL, if you receive a similar error message to this...

```
[Cloudera][ImpalaODBC] (110) Error while executing a query in Impala:
[HY000] : ExecQueryFInstances rpc query_id=id-number failed: Failed to
get minimum memory reservation of 9.94 MB on daemon
dn##.COMPANY.COM:22000 for query id-number because it would exceed an
applicable memory limit. Memory is likely oversubscribed. Reducing
query concurrency or configuring admission control may help avoid this
error.
```

...please work with your Hadoop Administrator as there may be some hardware issues that need attending to.

Name Node is in Safe Mode

If you get the following while running a SQL query...

```
org.apache.hive.service.cli.HiveSQLException: Failed to open new
session: java.lang.RuntimeException:
org.apache.hadoop.ipc.RemoteException(org.apache.hadoop.ipc.RetriableEx
ception): org.apache.hadoop.hdfs.server.namenode.SafeModeException:
Cannot create directory /tmp/hive/hive/8aa5f721-8ab8-42cd-adea-
e89486a25fd7. Name node is in safe mode.
```

...please contact your Hadoop Administrator immediately!

Error Converting Column

The following warning...

```
WARNINGS: Error converting column: # to data-type
Error parsing row: file: hdfs://directory/.../table-name, before
offset: ####
Error converting column: # to data-type
```

...usually inidicates that the underlying data file may have a field value that cannot be converted to the `data-type` specified for the column indicated by the pound sign (#) after the text **`Error converting column:`**. Depending

on the field values in that column's underlying data, you may be able to ignore this message. But, it's best to contact your Hadoop Administrator so he/she can ensure that the field from the underlying data has been mapped to the correct column in the table. For example, despite the data dictionary indicating a very specific format for a date field (say `YYYY-MM-DD`), I've seen several instances where some rows careen off into the nether regions of incomplete or malformed date formats. For example, you may see something similar to the following:

```
Parquet file 'hdfs://directory/.../table-name' column 'column-name'
contains an out of range timestamp. The valid date range is 1400-01-
01..9999-12-31.
```

Communication Link Failure

If you receive the following, or similar, message...

```
Caused by: com.cloudera.impala.support.exceptions.ErrorException:
[Cloudera][ImpalaJDBCDriver](500593) Communication link failure. Failed
to connect to server. Reason: Unknown.
```

...please contact your Hadoop Administrator indicating the approximate time you received this message.

Cannot Re-Acquire Authentication Token (Kudu Tables)

If you receive the following, or similar, message...

```
ERROR: ImpalaRuntimeException: Error creating Kudu table
'impala::prod_schema. table-name'
CAUSED BY: NonRecoverableException: cannot re-acquire authentication
token after 5 attempts
```

...please contact your Hadoop Administrator.

Server Requires Kerberos But Client Not Authorized

If you receive the following, or similar, message...

```
Unhandled exception in HPL/SQL
java.sql.SQLException: [Cloudera][ImpalaJDBCDriver](500051) ERROR
processing query/statement. Error Code: 0, SQL state:
TStatus(statusCode:ERROR_STATUS, sqlState:HY000,
errorMessage:ImpalaRuntimeException: Error creating Kudu table
'impala::prod_schema.table-name'
CAUSED BY: NonRecoverableException: Server requires Kerberos, but this
client is not authenticated (kinit)
CAUSED BY: SaslException: GSS initiate failed
CAUSED BY: GSSException: No valid credentials provided (Mechanism
level: Failed to find any Kerberos tgt)
```

...contact your Hadoop Administrator as he/she may need to restart the Kudu master service.

Rejected Query from Pool

If you're using request pools and you receive the following error message...

```
ERROR: Rejected query from pool pool-name: request memory needed ### GB
is greater than pool max mem resources ### GB.
```

...try to re-structure your SQL query. If the problem persists, contact your Hadoop Administrator who may modify the request pool to allow for more resources.

Admission for Query Exceeded Timeout

If you're using request pools and you receive the following error message...

```
ERROR: Admission for query exceeded timeout ###ms in pool pool-name.
Queued reason: number of running queries # is over limit #.
```

...the resource pool has been set to limit the number of concurrently running queries. Please contact your Hadoop Administrator to see if that limit can be increased, or submit your query to a different request pool or during off-peak hours.

Not Enough Live Tablet Servers (Kudu Tables)

If you receive the following error message when attempting to create a Kudu table...

```
[Cloudera][ImpalaJDBCDriver](500051) ERROR processing query/statement.
Error Code: 0, SQL state: TStatus(statusCode:ERROR STATUS,
sqlState:HY000, errorMessage:ImpalaRuntimeException: Error creating
Kudu table 'impala::prod_schema.table-name'
CAUSED BY: NonRecoverableException: Not enough live tablet servers to
create a table with the requested replication factor #. # tablet
servers are alive.)
```

...contact your Hadoop Administrator as one or more Kudu processes may need to be restarted.

Unable to Initialize the Kudu Scan Node (Kudu Tables)

If you receive the following error message when attempting to query a Kudu table...

```
ERROR: ImpalaRuntimeException: Unable to initialize the Kudu scan node
```

...your Hadoop Administrator may need to restart the Kudu service.

Memory Limit Exceeded

If you receive the following message when attempting to create a table...

```
WARNINGS: Memory limit exceeded: Cannot perform aggregation at node
with id 3. Failed to allocate 30 bytes for intermediate tuple.
Fragment ### could not allocate 30.00 B without exceeding limit.
Error occurred on backend dn###.COMPANY.COM:22000 by fragment ###
Memory left in process limit: ### GB
Memory left in query limit: ### KB
```

...your query may have exceeded the resource pool limits. Please contact your Hadoop Administrator to see if he/she can increase the pool limits. Note that this particular message can appear in other guises besides resource pools!!

Authorization Exception When Using BETWEEN Operator

If you receive the following message when querying or creating a table containing the BETWEEN Operator...

```
ERROR: AuthorizationException: User 'smithbob@COMPANY.COM' does not
have privileges to access: server2
```

Now, one way to…oh no!…I'm getting another text message…it's from my Father now…hold that thought…

It's your Father. I need you to take me to a pub.

You're married, so I'm not legally allowed to do that in this state. What's wrong now, Pa?

Same old story: your Mother is driving me crazy. And her eggplant parmesan tastes funny. Nevermind about that...how are you?

Fine. I'm having trouble getting a SQL query to run when I use the BETWEEN operator. I don't expect you to understand.

Have you tried replacing the BETWEEN operator with the <= or >= signs, or even the IN operator?

WHAT? WHAT? WHAT? How do you know that?

Your mother talks in her sleep.

Thanks, Pa! I'll give that a shot.

Good! So...uh...am I gettin' that beer?

Sure, Pa! Where do you want to go?

Antarctica?

Oy.

As the elderly Mr. Smith suggested, try replacing the `BETWEEN` Operator with an equivalent construct such as `<=` and `>=` or, even, the `IN` Operator.

Failed to Write Batch of # Ops to Tablet (Kudu Tables)

If you receive the following message when updating, querying or creating a Kudu table...

```
WARNINGS: Kudu error(s) reported, first error: Timed out: Failed to
write batch of 52 ops to tablet
```

...contact your Hadoop Administrator as the Kudu service may need to be restarted.

Client Connection Negotiation Failed

If you receive the following error message when creating a table...

```
ERROR processing query/statement. Error Code: 0, SQL state: Unable to
open scanner: Timed out: Client connection negotiation failed: client
connection to ###.###.###.###:7050: Timeout exceeded waiting to
connect: Network error: Client connection negotiation failed: client
connection to ###.###.###.###:7050: connect: Connection refused (error
111)
```

...contact your Hadoop Administrator.

Failed to Write Data to HDFS File

If you receive the following error message when creating a table...

```
WARNINGS: Failed to write data (length: 38080) to Hdfs file:
hdfs://directory/.../filename.parq
Error(255): Unknown error 255
Root cause: RemoteException: The DiskSpace quota of
/data/prod/teams/prod_schema is exceeded: quota = # B = ### but
diskspace consumed = # B = ### TB
```

...contact your Hadoop Administrator as the disk space in HDFS may need to be extended.

Disk I/O Error: Failed to Open HDFS File

If you receive the following error message...

```
Disk I/O error: Failed to open HDFS file hdfs://directory/.../file-name
Error(2): No such file or directory
Root cause: RemoteException: File does not exist: /directory/.../file-
name
```

...try to `INVALIDATE METADATA` the table and resubmit your query. If the error still persists, contact your Hadoop Administrator.

Error Validating LDAP User (HPL/SQL)

If you receive the following error message...

```
Unhandled exception in HPL/SQL
java.sql.SQLException: Could not open client transport with JDBC Uri:
jdbc:hive2://servername:10000/prod_schema: Peer indicated failure:
PLAIN auth failed: Error validating LDAP user
```

...the password of the user who submitted the HPL/SQL program may need to be reset. Contact your Hadoop Administrator.

Permission Denied When Using `distcp`

Although we talked about the `hadoop` utility earlier in the book, we didn't touch on the `distcp` command since it's rather esoteric. This command allows you to copy HDFS files from one Hadoop cluster to another Hadoop cluster…which assumes you have multiple clusters! In any case, if you're getting a permission denied error when using the `distcp` command, it may be caused by the two clusters having different usernames/passwords from each other. If you need tables copied from one cluster to another, contact your glorious Hadoop Administrator.

Appendage #5 – Where Do I Go from Here?

If you've made it this far without having an aneurysm, CONGRATULATIONS! In this appendage, we'll point you in the direction of some additional resources you can peruse to fill in some of the gaping holes left in this book.

- Websites
  - 7-Zip: `https://www.7-zip.org/`
  - Antlr Website: `https://www.antlr.org/`
  - Apache Hadoop: `https://hadoop.apache.org/`
  - Apache Hadoop Commands `https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/CommandsManual.html`
  - Apache Hive: `https://hive.apache.org/`
  - Apache HiveQL Reference: `https://cwiki.apache.org/confluence/display/Hive//LanguageManual`
  - Apache Impala: `https://impala.apache.org/`
  - Apache ImpalaSQL Reference: `https://impala.apache.org/docs/build/html/topics/impala_langref.html`
  - Apache Parquet: `https://parquet.apache.org/`
  - Apache Kudu: `https://kudu.apache.org/`
  - Apache Sqoop: `https://sqoop.apache.org/`
  - Apache Sqoop Reference Manual: `https://sqoop.apache.org/docs/1.4.7/SqoopUserGuide.html`
  - CloneZilla: `https://clonezilla.org/`
  - Cloudera: `https://www.cloudera.com/`
  - Crontab Guru: `https://crontab.guru/`
  - Cygwin: `https://www.cygwin.com/`
  - Docker: `https://www.docker.com/`
  - HPL/SQL: `http://www.hplsql.org/home`
  - HPL/SQL Reference Manual: `http://www.hplsql.org/doc`
  - Hortonworks: `https://www.cloudera.com/products/hdp.html`
  - Java: `https://www.java.com`
  - Macrium Reflect: `https://www.macrium.com/reflectfree`
  - MySQL Connectors: `https://www.mysql.com/products/connector/`
  - Parquet Viewer: `https://github.com/mukunku/ParquetViewer`
  - PeaZip: `https://www.peazip.org/`
  - Taco Bell World Domination Website: `https://www.tacobell.com/`
  - UpGuard VMWare vs. Docker: `https://www.upguard.com/blog/docker-vs-vmware-how-do-they-stack-up`
  - VirtualBox: `https://www.virtualbox.org`
  - VMWare: `https://www.vmware.com/`
- Books
  - Hadoop: The Definitive Guide by Tom White
  - Learning the vi and Vim Editors by Arnold Robbins and Elbert Hannah
  - Getting Started with Impala: Interactive SQL for Apache Hadoop by John Russel
  - Programming Hive: Data Warehouse and Query Language for Hadoop by Edward Capriolo, et. al.
  - Mastering Regular Expressions by Jeffrey E.F. Friedl
  - Learning the Bash Shell by Cameron Newham
  - Managing Projects with GNU Make by Robert Mecklenburg
  - Java: How to Program by Paul Deitel and Harvey Deitel
  - The Definitive ANTLR 4 Reference by Terence Parr
  - Why I Loathe Linux Administrators by Linus Torvalds
- Additional Items
  - Call your Mother.

Appendage #6 – ISO Latin-1 (8859-1) Character Set

[This chapter has been placed here purely to increase the page count because the book is so thin.]

Entity Number	Hex	Character	Entity Name	Description
&#000;	00	^@		null
&#001;	01	^A		start of heading
&#002;	02	^B		start of text
&#003;	03	^C		end of text
&#004;	04	^D		end of transmission
&#005;	05	^E		enquiry
&#006;	06	^F		acknowledge
&#007;	07	^G		bell
&#008;	08	^H		backpace
** **	09	^I		horizontal tab
**
**	0A	^J		line feed, new line
&#011;	0B	^K		vertical tab
****	0C	^L		form feed, new page
**
**	0D	^M		carriage return
&#014;	0E	^N		shift out
&#015;	0F	^O		shift in
&#016;	10	^P		data link escape
&#017;	11	^Q		device control 1
&#018;	12	^R		device control 2
&#019;	13	^S		device control 3
&#020;	14	^T		device control 4
&#021;	15	^U		negative acknowledge
&#022;	16	^V		synchonous idle
&#023;	17	^W		end of transmission block
&#024;	18	^X		cancel
&#025;	19	^Y		end of medium
&#026;	1A	^Z		substitute
&#027;	1B	^[		escape
&#028;	1C	^\		file separator
&#029;	1D	^]		group separator
&#030;	1E	^^		record separator
&#031;	1F	^_		unit separator
** **	20		&sp;	space
!	21	!	!	exclamation mark
"	22	"	"	double quotation mark
#	23	#	#	number sign, pound
$	24	$	$	dollar sign
%	25	%	%	percent sign
&	26	&	&	ampersand
'	27	'	'	apostrophe, single quote mark

(	28	(	(	left parenthesis
)	29	)	)	right parenthesis
*****	2A	*	*	asterisk
+	2B	+	+	plus sign
,	2C	,	,	comma
-	2D	-	− ‐	minus sign, hyphen
.	2E	.	.	period, decimal point, full stop
/	2F	/	/	slash, virgule, solidus
0	30	0		digit 0
1	31	1		digit 1
2	32	2		digit 2
3	33	3		digit 3
4	34	4		digit 4
5	35	5		digit 5
6	36	6		digit 6
7	37	7		digit 7
8	38	8		digit 8
9	39	9		digit 9
:	3A	:	:	colon
;	3B	;	;	semicolon
<	3C	<	<	less-than sign
=	3D	=	=	equal sign
>	3E	>	>	greater-than sign
?	3F	?	?	question mark
@	40	@	@	commercial at sign
A	41	A		capital A
B	42	B		capital B
C	43	C		capital C
D	44	D		capital D
E	45	E		capital E
F	46	F		capital F
G	47	G		capital G
H	48	H		capital H
I	49	I		capital I
J	4A	J		capital J
K	4B	K		capital K
L	4C	L		capital L
M	4D	M		capital M
N	4E	N		capital N
O	4F	O		capital O
P	50	P		capital P
Q	51	Q		capital Q
R	52	R		capital R

S	53	S		capital S
T	54	T		capital T
U	55	U		capital U
V	56	V		capital V
W	57	W		capital W
X	58	X		capital X
Y	59	Y		capital Y
Z	5A	Z		capital Z
[	5B	[	[	left square bracket
****	5C	\	\	backslash, reverse solidus
]	5D	]	]	right square bracket
^	5E	^	ˆ	spacing circumflex accent
_	5F	_	_ ―	spacing underscore, low line, horizontal bar
`	60	`	`	spacing grave accent, back apostrophe
a	61	a		small a
b	62	b		small b
c	63	c		small c
d	64	d		small d
e	65	e		small e
f	66	f		small f
g	67	g		small g
h	68	h		small h
i	69	i		small i
j	6A	j		small j
k	6B	k		small k
l	6C	l		small l
m	6D	m		small m
n	6E	n		small n
o	6F	o		small o
p	70	p		small p
q	71	q		small q
r	72	r		small r
s	73	s		small s
t	74	t		small t
u	75	u		small u
v	76	v		small v
w	77	w		small w
x	78	x		small x
y	79	y		small y
z	7A	z		small z
{	7B	{	{	left brace, left curly bracket
|	7C	|			vertical bar

}	7D	}	}	right brace, right curly bracket
~	7E	~	˜	tilde accent
&#127;	7F	^?		delete
€	80			
	81			
‚	82			
ƒ	83			
„	84			
…	85			
†	86			
‡	87			
ˆ	88			
‰	89			
Š	8A			
‹	8B			
Œ	8C			
	8D			
Ž	8E			
	8F			
	90			
‘	91			
’	92			
“	93			
”	94			
•	95			
–	96			
—	97			
˜	98			
™	99			
š	9A			
›	9B			
œ	9C			
	9D			
ž	9E			
Ÿ	9F			
** **	A0			non-breaking space
¡	A1	¡	¡	inverted exclamation mark
¢	A2	¢	¢	cent sign
£	A3	£	£	pound sterling sign
¤	A4	¤	¤	general currency sign
¥	A5	¥	¥	yen sign
¦	A6	¦	&brkbar; ¦	broken vertical bar
§	A7	§	§	section sign

¨	A8	¨	¨ ¨	spacing dieresis or umlaut
©	A9	©	©	copyright sign
ª	AA	ª	ª	feminine ordinal sign
«	AB	«	«	left double angle quote or guillemet
¬	AC	¬	¬	logical not sign
****	AD			soft hyphen
®	AE	®	®	registered trademark sign
¯	AF	¯	¯ &hibar;	spacing macron long accent
°	B0	°	°	degree sign
±	B1	±	±	plus-or-minus sign
²	B2	²	²	superscript 2
³	B3	³	³	superscript 3
´	B4	´	´	spacing accute accent
µ	B5	µ	µ	micro sign, mu
¶	B6	¶	¶	paragraph sign, pilcrow sign
·	B7	·	·	middle dot, centered dot
¸	B8	¸	¸	spacing cedilla
¹	B9	¹	¹	superscript 1
º	BA	º	º	masculine ordinal indicator
»	BB	»	»;	right double angle quote or guillemet
¼	BC	¼	¼	fraction 1/4
½	BD	½	½ ½	fraction 1/2
¾	BE	¾	¾	fraction 3/4
¿	BF	¿	¿	inverted question mark
À	C0	À	À	capital A grave
Á	C1	Á	Á	capital A acute
Â	C2	Â	Â	capital A circumflex
Ã	C3	Ã	Ã	capital A tilde
Ä	C4	Ä	Ä	capital A dieresis or umlaut
Å	C5	Å	Å	capital A ring
Æ	C6	Æ	Æ	capital AE ligature
Ç	C7	Ç	Ç	capital C cedilla
È	C8	È	È	capital E grave
É	C9	É	É	capital E acute
Ê	CA	Ê	Ê	capital E circumflex
Ë	CB	Ë	Ë	capital E dieresis or umlaut
Ì	CC	Ì	Ì	capital I grave
Í	CD	Í	Í	capital I acute
Î	CE	Î	Î	capital I circumflex
Ï	CF	Ï	Ï	capital I dieresis or umlaut
Ð	D0	Ð	Ð	capital ETH
Ñ	D1	Ñ	Ñ	capital N tilde

Ò	D2	Ò	Ò	capital O grave
Ó	D3	Ó	Ó	capital O acute
Ô	D4	Ô	Ô	capital O circumflex
Õ	D5	Õ	Õ	capital O tilde
Ö	D6	Ö	Ö	capital O dieresis or umlaut
×	D7	×	×	multiplication sign
Ø	D8	Ø	Ø	capital O slash
Ù	D9	Ù	Ù;	capital U grave
Ú	DA	Ú	Ú	capital U acute
Û	DB	Û	Û	capital U circumflex
Ü	DC	Ü	Ü	capital U dieresis or umlaut
Ý	DD	Ý	Ý	capital Y acute
Þ	DE	Þ	Þ	capital THORN
ß	DF	ß	ß	small sharp s, sz ligature
à	E0	à	à	small a grave
á	E1	á	á	small a acute
â	E2	â	â	small a circumflex
ã	E3	ã	ã	small a tilde
ä	E4	ä	ä	small a dieresis or umlaut
å	E5	å	å	small a ring
æ	E6	æ	æ	small ae ligature
ç	E7	ç	ç	small c cedilla
è	E8	è	è	small e grave
é	E9	é	é	small e acute
ê	EA	ê	ê	small e circumflex
ë	EB	ë	ë	small e dieresis or umlaut
ì	EC	ì	ì	small i grave
í	ED	í	í	small i acute
î	EE	î	î	small i circumflex
ï	EF	ï	ï	small i dieresis or umlaut
ð	F0	ð	ð	small eth
ñ	F1	ñ	ñ	small n tilde
ò	F2	ò	ò	small o grave
ó	F3	ó	ó	small o acute
ô	F4	ô	ô	small o circumflex
õ	F5	õ	õ	small o tilde
ö	F6	ö	ö	small o dieresis or umlaut
÷	F7	÷	÷	division sign
ø	F8	ø	ø	small o slash
ù	F9	ù	ù	small u grave
ú	FA	ú	ú	small u acute
û	FB	û	û	small u circumflex
ü	FC	ü	ü	small u dieresis or umlaut
ý	FD	ý	ý	small y acute

þ	FE	þ	þ	small thorn
ÿ	FF	ÿ	ÿ	small y dieresis or umlaut

Epilogue

HOBOKEN (AP NEWS) – HOBOKEN TECHNOLOGY GIANT **COMPANY, INC.** HAS RELEASED ITS QUARTERLY FINANCIAL STATEMENT INDICATING A HUGE INFLUX OF BUSINESS DUE MAINLY TO ITS STREAMLINED INFORMATION TECHNOLOGY DEPARTMENT HEADED BY HOBOKEN-NATIVE BOB SMITH. IN A STATEMENT TO THE PRESS, SMITH DOWNPLAYED HIS ROLE IN THE COMPANY'S SUCCESS POINTING TO THE USE OF THE HADOOP DATABASE SYSTEM BY OTHER TECHNOLOGY GIANTS, SUCH AS GOOGLE. DUE TO HIS RECENT SUCCESS, SMITH HAS BEEN PROMOTED TO CHIEF TECHNOLOGY OFFICER (CTO) OF **COMPANY, INC.**

IN RELATED NEWS, A **COMPANY, INC.** SPOKESPERSON ANNOUNCED THAT **MIKE THE SALES GUY** RESIGNED TODAY CITING THE DESIRE TO FIND A PROPER SURNAME.

IN UNRELATED NEWS, MRS. SMITH HAS BEEN ARRESTED FOR POSSESSION AND DISTRIBUTION OF COCAINE IN A FOOD PRODUCT. WHILE BEING LED AWAY BY POLICE, SHE WAS OVERHEARD SHOUTING, "IT'S GRATED CHEESE FOR MY HOMEMADE EGGPLANT PARMESAN! I GAVE SOME OUT TO MY NEIGHBORS! ASK THEM!" NEIGHBORS WERE TOO STONED TO COMMENT.

IN MORE UNRELATED NEWS, MR. SMITH WENT TO A PUB AND SCORED DESPITE THE OVERPOWERING AROMA OF FORMALDEHYDE WAFTING FROM HIS PERSON.

FINALLY, WE HOPE Y'ALL ENJOYED READING THIS BOOK AS MUCH AS THE AUTHOR HAD WRITING IT. BEST OF LUCK WITH YOUR PROJECT, POPPETS, AND STAY WELL!

SMOOCHIES AND HUGGIES,
SCOTT H.

** END TRANSMISSION END TRANSMISSION END TRANSMISSION **

Index


RIP
JAKE